APPLIED DATA RESEARCH, INC.

See AD 737 259

A1

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

FOURTH SEMI-ANNUAL TECHNICAL REPORT
(14 July 1971 - 13 January 1972)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

VOLUME II

Principal Investigator and Project Leader:
Robert E. Millstein Phone (617) 245-9540

ARPA Order Number	ARPA 1554
Program Code Number	0D30
Contractor:	Applied Data Research, Inc.
Contract No.:	DAHC04 70 C 0023
Effective Date:	13 January 1970
Amount:	$916,712.50

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 1554

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

FOURTH SEMI-ANNUAL TECHNICAL REPORT
(14 July 1971 - 13 January 1972)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

VOLUME II

CA-7202-1111

Principal Investigator and Project Leader:
Robert E. Millstein

Phone (617) 245-9540

TABLE OF CONTENTS

VOLUME II

CHAPTER I

CONTROL STRUCTURES IN ILLIAC IV FORTRAN

Part of our effort for the design and implementation of the ILLIAC IV FORTRAN compiler has been the design of an extended FORTRAN, called IVTRAN, which provides a suitable means of programming the ILLIAC. The extensions to standard FORTRAN are statement forms for expressing parallelism and data layout in an array memory. This chapter will describe these structures, but it is primarily concerned with the rationale which led to their creation.

The logical starting point is the hardware which makes up an ILLIAC IV quadrant [1,2]. Let us review the features of the machine and pick out unconventional parts which might be expected to affect language design. The computing hardware consists of a control unit (CU) and 64 processing units (PUs). Each processing unit consists of a processing element (PE) and a processing element memory (PEM) of 2K 64 bit words. A PE memory access requires 1 cycle, and a normalized floating point add instruction requires 5 cycles. A cycle requires 62.5 ns., so the basic data rate is 64 bits per 62.5 ns. per processor, or $65\text{x}10^9$ bits per second (bps) for a quadrant of 64 processors. An average execution time of 312.5 ns. per instruction per processor produces an instruction rate of 200 (3.1) million instructions per second (MIP) per quadrant (processor). All instructions are interpreted by the CU, which decodes each instruction and broadcasts, synchronously, sequences of microinstructions to each PE. That is, the CU interprets an instruction and then each PE, simultaneously, executes that instruction. One operand may be broadcast from the CU. Other operands are available to the PE from its own PEM or operating registers. In addition, PEs may be disabled for the execution of any given (sequence of) instructions. That is, any set of PEs can be (temporarily) turned off during the course of an instruction stream. Thus, if an add instruction is broadcast by the CU, a given PE may execute it (on local data) or ignore it. It is not, however, possible to execute a different instruction. The first major unconventional feature of ILLIAC design is that there is exactly one instruction stream for 64 processors which operate synchronously on local data streams.

The CU is able to perform some integer arithmetic, primarily for loop control and address calculation, but the major computing power resides in the PE. The PEs can perform a standard repertoire of fixed point, floating point, and logical computations. Certainly the computing capability of each PE is conventional and poses no difficulty in language design.

An ILLIAC quadrant has 128K of memory, all of which is accessible, in conventional fashion, to the CU. Each PE, however, sees only 2K of local memory. The memory structure is depicted in Figure 1. Instructions which reference memory generate an effective address between 0 and 2047_{10}. This address is used as a displacement in each PEM. For instance, an instruction with effective address 100 would cause the 64 words marked by an asterisk in Figure 1 to be referenced. Each PE contains a local index register which can be used to modify the virtual address field of an instruction. Thus, if the index register of PE0 contained 0, the index register of PE1 contained 1, etc., and the virtual address field contained 100, the 64 words marked by a plus in Figure 1 would be referenced. Note, however, that a PE can only reference words within its PEM. The ILLIAC quadrant then, contains powerful processors which have conventional access to only 2K of memory. A routing instruction is provided to allow data transfers between PEs. The PEs are, in effect, connected in a closed circular fashion. The routing instruction transmits a word from each PE to the PE located n positions distant around the ring. A total of 64 words are transmitted: PE0 sends a word to PEn, PE1 sends to PE((n+1)mod64),... and PE63 sends to PE((n+63)mod64), $0 \le n \le 63$. As many as 64 of the transfers may be useful, or as few as 1, depending on data layout. Clearly, the routing instruction is insufficient to compensate for the small size of the memory directly addressable by each PE. This is the second major unconventional feature of the ILLIAC IV: a very small,locally accessable memory with minimal inter-memory connections.

Just as 2K is a small memory for a 3.1 MIP processor, 128K is a small memory for a 200 MIP machine. To compensate for the small overall memory a 16 million word disk file with a $.5 \times 10^9$ bps transfer rate is part of the ILLIAC quadrant. This enables a complete memory load to be accomplished in 16 ms. (ignoring latency) and provides a throughput access time per word of 128 ns., or approximately 1/130 PEM speed. This access time is clearly an idealization, but the extremely rapid transfer rate does alleviate to some extent the memory size constraints. This disk, however, rotates at conventional speeds and, hence, suffers from a 20 ms. latency. Since 20 ms. is enough time for 4 million instruction executions, this problem is a serious one. This conventional disk latency, coupled with an unconventionally high transfer rate, comprises the third pertinent ILLIAC hardware feature.

These, then, are the three crucial machine characteristics which influenced our language design:

- a single instruction stream controlling synchronous operation on 64 different data streams.
- a powerful, fast processor with only limited (2K) local memory and only limited connections (via routing instructions) to other high speed memory.
- a total of only 128K of program and data primary storage for a 200 MIP processor with a backup store characterized by high transfer rate and (relatively) slow access rate.

Note that every one of these hardware features concerns data. This concern is crucial to the problem of using the ILLIAC IV efficiently. The problem of optimizing for the ILLIAC -- by which we mean both the creation of efficient algorithms and the generation of efficient code -- is in large part concentrated on optimization of data organization rather than op'imization of machine code sequences. The ILLIAC will only operate effectively if data is laid out properly. Reorganizing data so that an operation can be performed on 64 items simultaneously is clearly a more important goal than utilizing index registers efficiently in a sequential instruction stream. This is not to say that conventional, code-oriented, optimization techniques are not applicable to the IVTRAN compiler; but, such techniques are of secondary importance to the optimization of data organization. In contrast to conventional computers, the optimization problem for the ILLIAC is one of data layout, not code reorganization.

Let us reexamine the hardware features listed above to see what conditions they suggest for ILLIAC optimization.

- The first feature -- a single instruction stream operating on 64 different data streams -- suggests that efficient ILLIAC code will be compiled from program structures where a single code sequence is executed for many different sets of data. In the case of FORTRAN, DO loops are clearly such program structures. In fact, the part of our effort related to the detection of parallelism in standard FORTRAN is concentrated on DO loops. Ignoring the question of data dependency, (Obviously, consideration of intra- and inter-loop data dependencies is of paramount importance to the problem of parallelism detection, but we can ignore it here.) a DO loop can be regarded as a single instruction stream operating on different data streams only if each data stream is represented as a variant of a single syntactic skeleton. That is, each data stream is a syntactic form that is distinguished by an index value. In FORTRAN, the only applicable form is the array. Therefore, the first ILLIAC hardware feature implies that the structuring of arrays appearing within DO loops will be crucial to efficient machine utilization.

- The second feature -- 2K of local memory with limited local memory-local memory transfer capability -- suggests that the crucial data structures -- i.e., arrays -- be allocated so that elements appearing in parallel data streams be located in different PEs. Then each PE, which is constrained to directly access only its local memory, will have direct access to the data required for computation. A secondary implication is that data be allocated so that memory-memory transfers -- accomplished by routing instructions -- move as many "good" data words as possible. That is, a routing instruction moves 64 words at a time; each word moves n PEs to the right, end around. If PEi requires a word in PE(2i mod 64), $0 \le i \le 63$, it can require 63 routine instructions (ignoring various clever algorithms for this case) to allocate data in appropriate PEs for use by perhaps only one instruction. In this example, each routing instruction moved only one "good" data word. Routing instructions require, on the average, 562.5 ns.; they can transmit as many as 4096 bits or as few as 64 bits. Thus, the data transfer rate can vary from as high as 7.3 x 10^9 bps to as low as .11 x 10^9 bps. Local memory accesses transfer bits at a rate of 65 x 10^9 bps, so it is important to keep routing instructions at the high end of the scale to preserve some semblance of a balance. (In fact, this discussion has ignored the fact that routing instructions are register to register operations and require an initial local memory access. This access pushes the bit rates into even greater imbalance.)

So far, we have seen that the first two unconventional ILLIAC hardware characteristics suggest certain conditions for data layout:

- arrays are crucial data objects.
- arrays should be allocated so that parallel data streams lie in different PEs.
- arrays should be allocated so that routing instructions move as many "good" data words as possible.

The third hardware feature basically concerns global data layout. This problem is not unique to the ILLIAC, and, further, is extremely sensitive to operating environment as well as machine characteristics. Hence, we will not treat it further in this chapter. Let us now examine a data allocation method which satisfies the above local conditions. Once we have examined this method it will be clearer how to provide control and allocation structures that let a programmer describe code which meets the local data layout restrictions imposed by the ILLIAC hardware.

Let us define a cross section of an array as the subarray obtained by holding some indices fixed and allowing all other indices to range through all permissible values. We recognize a special type of cross section, called a slice, which is the vector obtained by fixing n-1 indices of an n-dimensional array. These data objects are the obvious generalizations of the array element, the subarray obtained by fixing all indices. The intent of these definitions is that a cross section of an array represents exactly a cross section of parallel data streams. A natural requirement, therefore, is that an array be stored so that any given cross section be accessible in parallel -- i.e., lies across the PEs. Of course, cross sections with more than 64 elements can only be accessed 64 elements at a time.

A natural operation to perform on a cross section is to combine it, element by element, with another cross section of the same size. It is not feasible to require that every array cross section exactly line up (in the PEMs) with every other cross section of identical size. However, we can require that cross sections be separated by at most a single uniform route. By "single uniform route" we mean that a single routing instruction will align the elements of one cross section with the elements of any other cross section. Again, cross sections of more than 64 elements can only be aligned 64 elements at a time.

As implied by the two previous requirements, arrays with dimensions greater than 64 must be allocatable by any acceptable storage method. In addition, it is desirable that cross sections and individual elements be accessible via simple and general formulae, with as much calculation as possible done either at compile time or else simultaneously in the PEs at run time.

The storage schema we devised to satisfy these requirements is called physical skewing [3,4]. For simplicity, we will describe it in terms of slices. (In fact, the method satisfies generalizations of the above requirements for slices. In addition, slices are expected to be by far the most used cross sections.) Consider Figure 2. This depicts the storage of a 3 x 7 x 10 array, each element denoted by its subscript. As in Figure 1, we view the ILLIAC memory as 2-dimensional, with the PEs running horizontally and the individual PEMs running vertically. Thus, the element 111 is in word 0 of PE0, etc. The reason for the name "physical skewing" is obvious. A slice along each of the three possible axes is marked in the figure. The slice along the first dimension is a 3-vector whose elements are enclosed by diamonds. The 7-vector along the second dimension has its elements enclosed by rectangles. Finally, the 10-vector along the third dimension has its elements in circles.

Note first that for each slice each element is in a different PE than any other element of that slice. Hence, each slice can be accessed in parallel. Second, there is a horizontal spacing of 1 between elements of a slice. Given any element, the next element is in the PE immediately to the right; so, an n-element slice occupies n consecutive PEs. Hence, a single routing command can move any slice into any other n consecutive PEs so as to be aligned with any other n-element slice. For example, a single route of -4 would move the vector 151, 152,... into PEs 0 through 9 so that corresponding elements of that vector and the vector 111, 112, ... would be in the same PE. Finally, given the first element of a slice, succeeding elements can be located by moving 1 PE to the right and then down 7, 1, or 0 memory locations depending on whether the slice is along the first, second, or third dimensions. (7 is merely the size of the second dimension, which determines the height of each 2-dimensional subarray.) Hence, accessing slices can be done by a simple uniform procedure.

This method meets three of the four requirements, but the example had no dimensions greater than 64. Let us now see how the physical skewing method can be adapted to meet the fourth requirement. Suppose that we had to store this 3 x 7 x 10 array in a 4-PE machine. We can slice the array vertically into 4-element wide strips, as indicated in the figure, and stack the strips on top of each other so that, for instance, the element 115 would be in PE0, word 21; the element 119 in PE0, word 42, and so on. Elements of a slice can still be accessed in parallel, but only 4 elements at a time of course. The routing remains the same since routing left from PE0 wraps around to PE3 and conversely for routing right from PE3. The access formulae are slightly more complicated. When the succeeding element in a slice is 1 PE to the right of PE3, then we wrap around to PE0 to locate the correct PE, and we must also add 21, the product of the sizes of the first and second dimensions to whatever number of locations down we would previously have used. For example, the first 4 elements of the slice 214, 224, ... are located as before. To locate 254 we rotate back to PE0, go down 21 locations because of the wrap around and then go down 1 more location, because this is a slice along the second axis. With only a moderate increase in the complexity of the access formulae, this method does indeed satisfy the fourth requirement.

Unfortunately, it also wastes an immenso amount of space, so we devised a method of packing the array. Consider Figure 3. This represents the same array as before, with the same slices marked, packed into the memory of a 4-PE machine. We obtained the packing by overlapping the first and fourth, and the second and fifth 4-element wide strips. Parallel access and routing remain as before. The access formulae are again slightly complicated. Just as locating the proper PE is done modulo 4, now locating the memory location is done modulo 63, the number of rows of memory that the matrix occupies. For example, the first 4 elements of the slice 151,152,... are located as before by moving 1 PE to the right and 0 elements down. The next 4 elements are located by wrapping around to PE0 and counting down 21. The remaining 2 elements are located by again wrapping around to PE0 and again counting down 21, which, modulo 63, brings us around to the top of the storage area. This method still leaves some wasted space, but the amount wasted is not impossibly large. Further, that space can be used for the storage of various compile-time-calculable constants, such as the product of dimension sizes, which are required by the access formulae, as well as for the storage of other constants required by the program.

Now this storage scheme

- treats arrays as the crucial data object
- allocates parallel data streams into different PEs
- maximizes economical routing

so it seems to satisfy the data layout conditions imposed by the ILLIAC hardware. We can therefore, introduce a statement to express parallel computations since we have an acceptable way of storing data for such computations. In addition to ordinary DO loops (for expressing sequential calculations), IVTRAN has the DO FOR ALL loop.

A DO FOR ALL statement is of the form:

DO k FOR ALL $(i_1, \ldots, i_n)/s$

where:

1) k is the statement label of an executable statement. This statement, called the terminal statement of the associated DO FOR ALL, must physically follow and be in the same program unit as the DO FOR ALL statement.

2) Each i_j is an integer scalar variable name called a control index. $(i_1,\ldots i_n)$ is called a control multi-index.

3) s is an n-dimensional logical array expression with an extent d.

Associated with the DO FOR ALL is a range defined to be those executable statements following the DO FOR ALL from and including the first statement following the DO FOR ALL to and including the terminal statement. We also define the extended range by replacing (by body substitution) function and subroutine calls by the referenced subprograms. This substitution is repeated iteratively until all such calls have been replaced.

A DO FOR ALL statement is used to specify that certain assignment statements within the extended range are to be executed for a set of values in parallel. This set of values, called the index set, is defined to be the set of n-tuples of integers $(i_1,\ldots,i_n)$ such that $s(i_1,\ldots,i_n)$ is true and $(i_1,\ldots,i_n) \in d$.

The extended range of a DO FOR ALL may not contain another DO FOR ALL or any array expressions. The control indices may not be used in any statement within the extended range of the DO FOR ALL loop, except as outlined below. Otherwise, any statement permitted elsewhere in the procedure part of the program is permitted and has its usual interpretation.

Within a DO FOR ALL extended range, the control indices may appear only in DO FOR ALL assignment statements. A DO FOR ALL assignment statement is one of the following forms:

p=e

or

IF(f)p=e

where:

1) p is an array element reference with subscripts of the form:

I

or I+C

or I-C

where I is one of the i_j and C is an expression independent of the i_j. Further, if i_j and i_k both appear as subscripts, then $j \neq k$ - i.e., the same control index cannot appear in more than one coordinate position.

2) e and f are each expressions which may or may not depend upon $(i_1,\ldots,i_n)$. Within e and f, any array references are either of the form 1) above or are independent of the i_j.

Execution of a DO FOR ALL assignment statement causes parallel assignment of the expression e to the array element reference p for all values $(i_1,\ldots,i_n)$ in the index set. The computation of the expressions e and f makes use of values of p in effect immediately before execution of the statement. If the second form of the statement is used, assignment is made only for those values of $(i_1,\ldots,i_n)$ in the index set for which $f(i_1,\ldots,i_n)$=.TRUE.

For example, let A(I,J,K) be the array shown in Figure 2. The following loop sets each element of A to its absolute value and then replaces that value by its square root.

```
      DO 1 FOR ALL (I,J,K)/[1...3].C.[1...7].C.[1...10]
      IF (A(I,J,K).LT.0.0) A(I,J,K)= -A(I,J,K)
  1   A(I,J,K)=SQRT(A(I,J,K))
```

.C. is an abbreviation of .CROSS., the Cartesian cross product operator. The built in square root function computes 64 values simultaneously, one in each PE.

We now note that it is possible, within the data storage scheme, to allocate a given array in several different ways. Consider the previous example; the array was stored so that vectors along the third dimension (elements enclosed by a circle in Figures 2 and 3) lie on horizontal lines. That is, the same effective address, computable at compile time and broadcast from the CU, locates each element of the slice. No PE indexing is required as it is for slices along the first and second dimensions. This allocation is most efficient if slices along the third dimension are accessed most often. However, if slices along either the first or second dimensions are most often accessed, then the array should be rotated so that, for example, slices along the second dimension lie on horizontal lines (see Figure 4). Thus, for maximum efficiency, it is necessary that the programmer specify the precise allocation that he desires. Furthermore, physical skewing allows parallel access to any slice along any dimension. If only slices along a single dimension need be accessed in parallel, then a simpler storage scheme is more efficient. This scheme is the normal "block" arrangement for arrays, with the parallel dimension stored row-wise. This scheme can be incorporated within physical skewing to permit highly efficient parallel access along some dimension, no parallel access along the "perpendicular" dimension and physically skewed parallel access along yet another dimension. Figure 5 depicts a 3 x 7 x 10 array stored this way. In any event, many variations are possible in the allocation of any given array. Hence, we require a statement form that allows a programmer to precisely specify the desired allocation. The allocation declaration provides this form.

The occurrence of an array variable in a DIMENSION, Type, or COMMON statement may be accompanied by an (optional) allocation declaration. Each variable may have no more than one such declaration in any program piece. The syntax of an allocation is

allocation :: =

[multi-index $\{\text{,multi-index}\}_o^\infty$]

multi-index :: =

(index$\{\text{ ,index}\}_o^\infty$) |

aligned (index$\{\text{ ,index}\}_o^\infty$ |

preferred (index$\{\text{ ,index}\}_o^\infty$

index :: = integer constant

aligned :: = #

preferred :: = $

An allocation is used to describe a desired storage map. Each index denotes a subscript position (e.g., index 2 denotes the second subscript position of an array A(I,J,K)). The order of indices within a multi-index is significant, but the order of multi-indices within an allocation is not.

If an index is preferred, then an increment of 1 in the index value will increase the PE number by 1, but the row number will not change. If an index is aligned, then an increment of 1 in the index value will not change the PE number, but the row number will change. We will further describe the allocation declaration by means of a series of examples. Consider a 3 x 5 array A(I,J). The allocation for storing this array physically skewed is [(1),(2)], which is also the default allocation (see Figure 6a). This allocation can also be written [(1),$(2)], indicating that subscript position 2 is the preferred index. This allocation permits parallel access along either coordinate, but access to the second coordinate is less expensive. It might be used to store an array A(I,J) which often appears in DO FORALL J loops and occasionally appears in DO FORALL I loops. The opposite case -- many DO FORALL I loops and few DO FORALL J -- suggests the transpose storage map, obtained by the allocation [$(1),(2)] (Figure 6b). If only slices along the second coordinate need be accessed in parallel, the aligned storage map (Figure 6c) can be obtained by the allocation [#(1),(2)]. If the array appears in the body of a DO FORALL I,J loop, then the most desirable storage map permits access to I,J cross-sections (Figure 6d). The allocation [(1,2)] produces this storage map. In general, multi-indices are used to obtain parallel access to cross-sections. (In the previous examples, the cross-sections were 1-dimensional slices, so the multi-indices consisted of only 1 index.) In all cases, slicing into 64 element wide strips is done automatically and is transparent to the user. As a final example, the allocation [(1),$(2),(3)] will produce the storage map in Figure 4.

Now an allocation, unfortunately, is not necessarily of the same efficiency throughout an entire program. The parallel data streams may be along one dimension in one loop and along a different dimension in another loop. It is up to the programmer to decide whether it is more efficient to change allocations between loops, or to select a compromise allocation. For this reason it is necessary to allow allocation declarations to be associated not just with program pieces but with loops.

A programmer can associate an allocation for an array A with a loop (or other program part) by the following technique:

- Define a dummy array A' with the desired allocation and the same extent as A
- Substitute A' for A in the loop
- Insert A'=A before the loop

The compiler will automatically call a subroutine to transform A into A'. This technique requires, of course, that twice as much space be given for the storage of A as is strictly necessary (since two different forms of the array are simultaneously defined). Often, such waste is not allowable. Further, the original allocation of A may not be needed again. Hence, it is desirable that a means be provided to reallocate A into its original storage area. The OVERLAP statement provides such a means.

The OVERLAP statement is of the form:

OVERLAP $(s_1, s_2, \ldots, s_n)$

where each s is an OVERLAP specifier of the form:

$(e_1, e_2, \ldots, e_m)$

and each e is an OVERLAP element of the form:

$(n_1, n_2, \ldots, n_k)$

and each n is an array or scalar variable name.

Either all of the variables in an OVERLAP specifier must be in the same COMMON block or none of them may be in COMMON. The order in which variables appear in OVERLAP elements is arbitrary. The order in which OVERLAP elements appear in OVERLAP specifiers is arbitrary. The same variable or array name may appear at most once in an OVERLAP statement.

Each OVERLAP specifier indicates sets of variables (overlap elements) which the compiler may cause to share storage. Each of the variables in an OVERLAP element can share storage with any of the variables in any other OVERLAP element in the same OVERLAP specifier. Variables declared in the same OVERLAP element do not share storage with one another.

If, in the previous example, the programmer wishes to conserve space, then he can use the statement:

```
OVERLAP ((A),(A'))
```

The compiler will then automatically reallocate A (and rename the result A') into the same storage area when the assignment A'=A is encounterea.

As a final example using these language features, we present IVTRAN code which reallocates a 3 x 5 array from the form shown in Figure 6a to the form in Figure 6d, and then, simulteneously, multiplies each element by π and computes the sine.

```
   DIMENSION A(3,5) [(1),(2)] A1(3,5) [(1,2)]
   OVERLAP ((A),(A1))
   A1 = A
   DO 1 FORALL (I,J)/[1...3].C.[1...5]
1  A1(I,J)=SIN(3.1416*A1(I,J))
```

The allocation declaration and OVERLAP statement, and the techniques for using them to change allocations, provide the programmer with the ability to adjust his data layout to best suit his algorithm. By using these statements together with DO FOR ALL loops, a programmer may precisely express both the parallel computations and the associated local data layout that utilizes the unconventional ILLIAC hardware features.

```
11 12 13 14 15
   21 22 23 24 25
      31 32 33 34 35
```

```
11 21 31
   12 22 32
      13 23 33
         14 24 34
            15 25 35
```

```
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
```

```
11 12 13 14 15 21 22 23 24 25 31 32 33 34 35
```

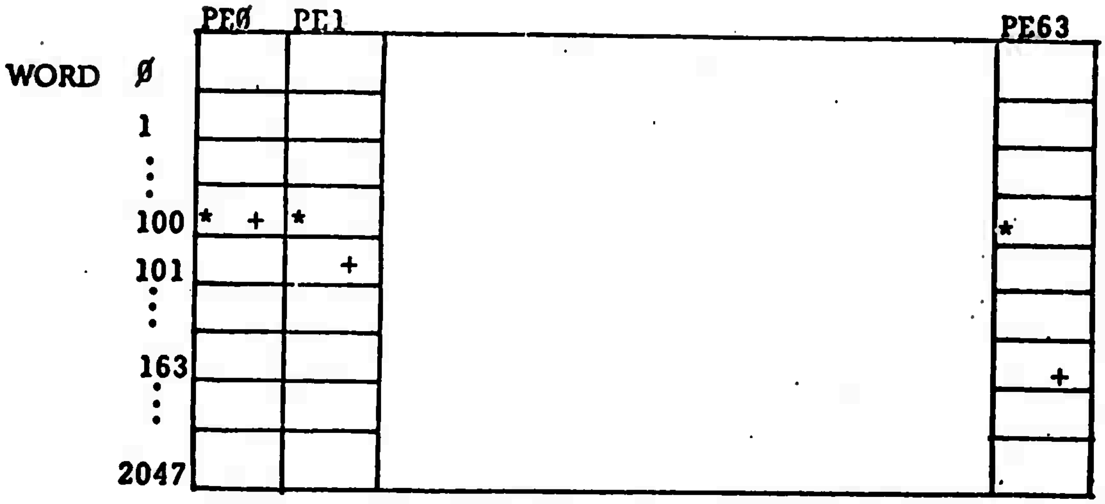
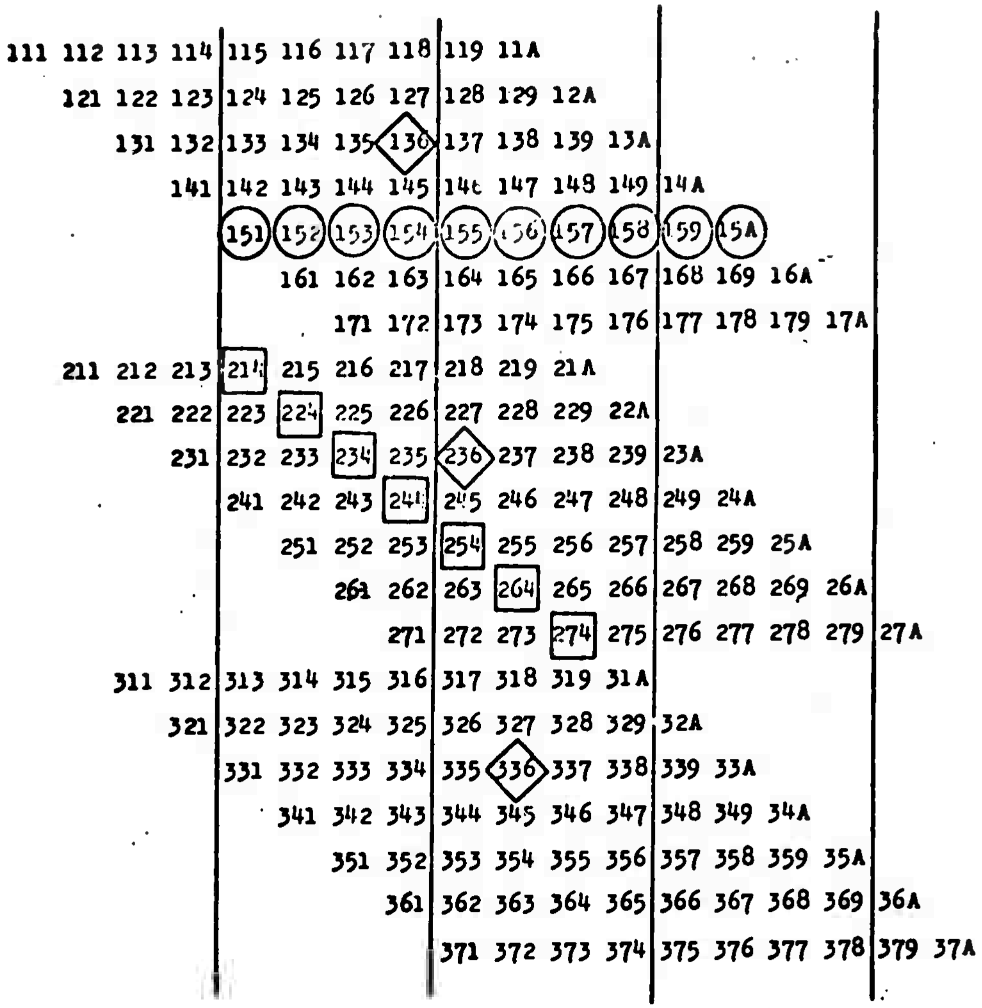


Figure 1



111 112 113 114 115 116 117 118 119 11A

121 122 123 124 125 126 127 128 129 12A

131 132 133 134 135 136 137 138 139 13A

141 142 143 144 145 146 147 148 149 14A

151 152 153 154 155 156 157 158 159 15A

161 162 163 164 165 166 167 168 169 16A

171 172 173 174 175 176 177 178 179 17A

211 212 213 214 215 216 217 218 219 21A

221 222 223 224 225 226 227 228 229 22A

231 232 233 234 235 236 237 238 239 23A

241 242 243 244 245 246 247 248 249 24A

251 252 253 254 255 256 257 258 259 25A

261 262 263 264 265 266 267 268 269 26A

271 272 273 274 275 276 277 278 279 27A

311 312 313 314 315 316 317 318 319 31A

321 322 323 324 325 326 327 328 329 32A

331 332 333 334 335 336 337 338 339 33A

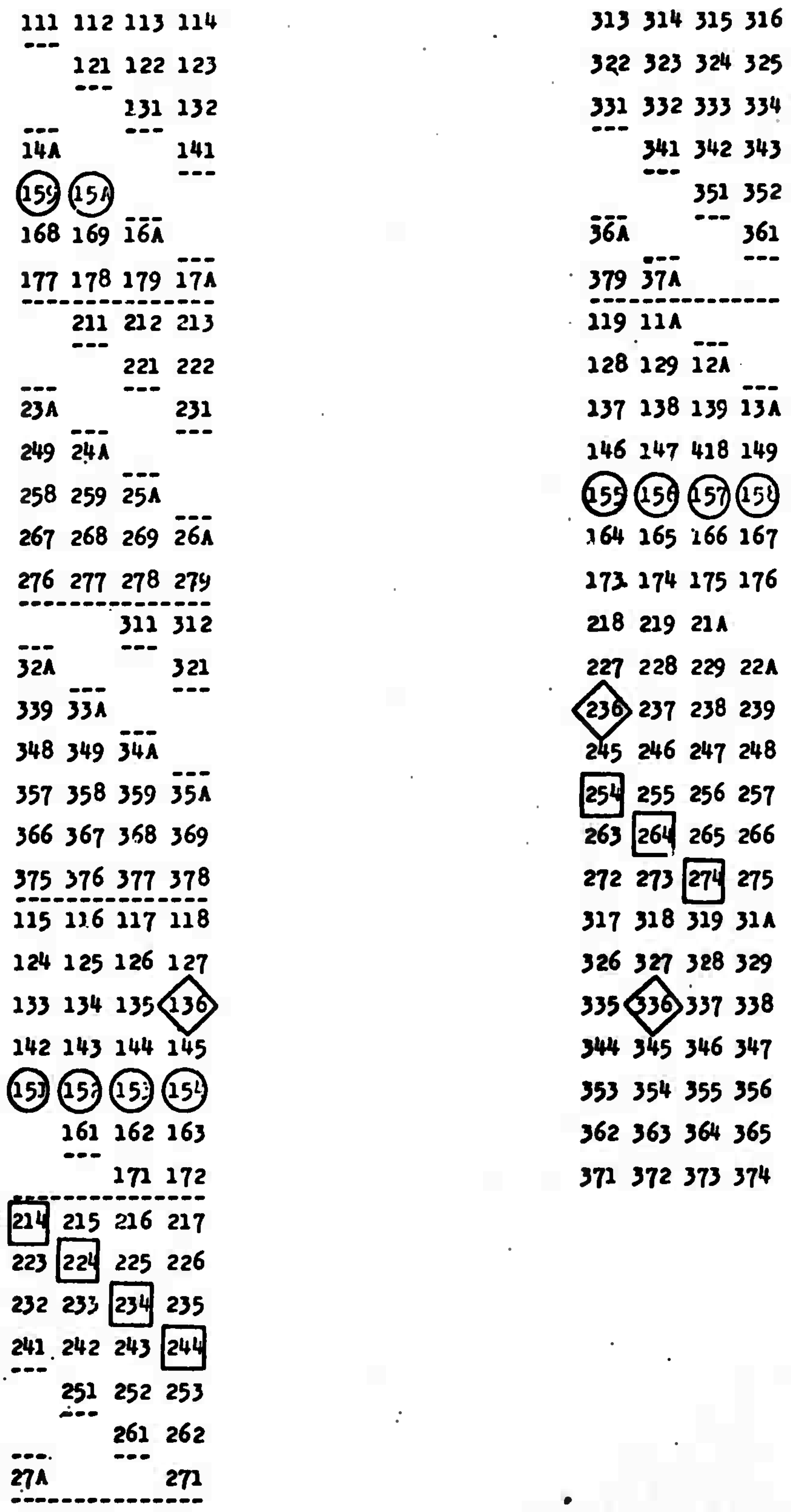
341 342 343 344 345 346 347 348 349 34A

351 352 353 354 355 356 357 358 359 35A

361 362 363 364 365 366 367 368 369 36A

371 372 373 374 375 376 377 378 379 37A

Figure 2



```
111 112 113 114
---
    121 122 123
    ---
        131 132
---     ---
14A         141
            ---
159 15A
        ---
168 169 16A
            ---
177 178 179 17A
---------------
    211 212 213
    ---
        221 222
---     ---
23A         231
    ---     ---
249 24A
        ---
258 259 25A
            ---
267 268 269 26A
276 277 278 279
---------------
        311 312
---     ---
32A         321
    ---     ---
339 33A
        ---
348 349 34A
            ---
357 358 359 35A
366 367 368 369
375 376 377 378
---------------
115 116 117 118
124 125 126 127
133 134 135 136
142 143 144 145
151 152 153 154
    161 162 163
    ---
        171 172
---------------
214 215 216 217
223 224 225 226
232 233 234 235
241 242 243 244
---
    251 252 253
    ---
        261 262
---     ---
27A         271
---------------
```

```
313 314 315 316
322 323 324 325
331 332 333 334
---
    341 342 343
    ---
        351 352
---     ---
36A         361
    ---     ---
379 37A
---------------
119 11A
        ---
128 129 12A
            ---
137 138 139 13A
146 147 418 149
155 156 157 158
164 165 166 167
173 174 175 176
218 219 21A
227 228 229 22A
236 237 238 239
245 246 247 248
254 255 256 257
263 264 265 266
272 273 274 275
317 318 319 31A
326 327 328 329
335 336 337 338
344 345 346 347
353 354 355 356
362 363 364 365
371 372 373 374
```

Figure 3

```
111 121 131 141 151 161 171
    112 122 132 142 152 162 172
        113 123 133 143 153 163 173
            114 124 134 144 154 164 174
                115 125 135 145 155 165 175
                    116 126 136 146 156 166 176
                        117 127 137 147 157 167 177
                            118 128 138 148 158 168 178
                                119 129 139 149 159 169 179
                                    11A 12A 13A 14A 15A 16A 17A
    211 221 231 241 251 261 271
        212 222 232 242 252 262 272
            213 223 233 243 253 263 273
                214 224 234 244 254 264 274
                    215 225 235 245 255 265 275
                        216 226 236 246 256 266 276
                            217 227 237 247 257 267 277
                                218 228 238 248 258 268 278
                                    219 229 239 249 259 269 279
                                        21A 22A 23A 24A 25A 26A 27A
        311 321 331 341 351 361 371
            312 322 332 342 352 362 372
                313 323 333 343 353 363 373
                    314 324 334 344 354 364 374
                        315 325 335 345 355 365 375
                            316 326 336 346 356 366 376
                                317 327 337 347 357 367 377
                                    318 328 338 348 358 368 378
                                        319 329 339 349 359 369 379
                                            31A 32A 33A 34A 35A 36A 37A
```

Figure 4

111 112 113 114 115 116 117 118 119 11A

121 122 123 124 125 126 127 128 129 12A

131 132 133 134 135 136 137 138 139 13A

141 142 143 144 145 146 147 148 149 14A

151 152 153 154 155 156 157 158 159 15A

161 162 163 164 165 166 167 168 169 16A

171 172 173 174 175 176 177 178 179 17A

211 212 213 214 215 216 217 218 219 21A

221 222 223 224 225 226 227 228 229 22A

231 232 233 234 235 236 237 238 239 23A

241 242 243 244 245 246 247 248 249 24A

251 252 253 254 255 256 257 258 259 25A

261 262 263 264 265 266 267 268 269 26A

271 272 273 274 275 276 277 278 279 27A

311 312 313 314 315 316 317 318 319 31A

321 322 323 324 325 326 327 328 329 32A

331 332 333 334 335 336 337 338 339 33A

341 342 343 344 345 346 347 348 349 34A

351 352 352 354 355 356 357 358 359 35A

361 362 363 364 365 366 367 368 369 36A

371 372 373 374 375 376 377 378 379 37A

Figure 5

Allocation Declaration

```
 11 12 13 14 15
    21 22 23 24 25
       31 32 33 34 35
[(1),(2)] or [(1),$(2)]
```

```
11 21 31
   12 22 32
      13 23 33
         14 24 34
            15 25 35
      [$(1),(2)]
```

```
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
      [#(1),(2)]
```

```
11 12 13 14 15 21 22 23 24 25 31 32 33 34 35
                 [(1,2)]
```

Figure 6

CHAPTER II

IVTRAN: A DIALECT OF FORTRAN FOR USE ON THE ILLIAC IV

1. INTRODUCTION

This manual describes the IVTRAN*language, a dialect of FORTRAN for use on the ILLIAC IV computer. The IVTRAN language has been designed both for use in converting existing FORTRAN programs for use on the ILLIAC IV and for construction of new programs to make use of the unique features the ILLIAC IV provides. The name "IVTRAN" is used to refer both to the language described in this manual and also to the language processor, called the compiler, which translates IVTRAN language programs into ILLIAC IV machine language.

1.1 The Language

To aid in program conversion and programmer training, the IVTRAN language incorporates many of the extensions to ANS standard FORTRAN which are part of the IBM/360 and CDC/6600 FORTRAN languages. These features are outlined in Appendix C.

A major feature of the IVTRAN system is the PARALYZER, a compiler option for use in program conversion. The PARALYZER examines DO-loops of a program to be converted for use on the ILLIAC IV and transforms the DO-loops into equivalent but more efficient DO FOR ALL statements which exploit the parallelism of the ILLIAC IV hardware. This feature is described in Appendix D.

The writer of new programs in IVTRAN has several features to aid in producing efficient programs. The first and most important of these features is the DO FOR ALL loop which specifies parallel operations on aggregates of data. Array expressions and statements provide a natural shorthand for certain commonly used parallel operations. Storage allocation for data can be specified so as to achieve the most effective compromise between storage efficiency and the use of parallelism. Storage allocation considerations are described in detail in Appendix B. Lastly, the user can aid

* pronounced "four-tran"

the compiler in optimizing his program by specifying expected execution frequencies through the FREQUENCY statement and external procedure side-effects through the EXTERNAL statement.

1.2 The Compiler and Linkage Editor

The IVTRAN compiler, operating on the TENEX version of the PDP-10, translates IVTRAN source program units into relocatable object modules. The linkage editor combines one or more modules translated by the compiler with any required library programs to form a load module which can be run upon the ILLIAC IV.

1.3 The Manual

This manual is organized to combine both conciseness and readability. Each language feature has its written form described in the body of the manual in English. Appendix A gives a formal syntax for the complete IVTRAN language. The meaning of each language feature is given in English in the body of the manual. Each section has examples of both valid and invalid uses of the language features described.

Although this manual is a complete description of the IVTRAN language, it can be more easily understood if the reader is already acquainted with the FORTRAN language for another computer. A useful source for this information is the set of programmed instruction texts produced by IBM, "FORTRAN IV for IBM System 360", IBM Order Nos. SR29-0080 through SR29-0087.

2. ELEMENTS OF IVTRAN

This chapter identifies the major structures of the IVTRAN language and serves as an introduction to the subsequent chapters.

2.1 Programs and Subprograms

A IVTRAN program consists of a main program or a main program plus one or more subprograms. Program execution begins at the first executable statement of the main program. Thereafter, unless a control statement is encountered, control proceeds from one executable statement to the next. Control statements cause control to transfer to another point in the same program unit or to an entry point of a subprogram.

There are three types of subprogram: the subroutine subprogram, which is referenced in a CALL statement; the function subprogram, which is referenced within an expression and returns a value for use in that expression; and the block data subprogram, which specifies initial data values for COMMON variables.

In IVTRAN, as with other FORTRAN systems, each program unit is compiled independently allowing a single program unit to be updated without requiring recompilation of other program units. However, unlike other FORTRAN systems, the linkage editor checks the consistency of specifications across program unit boundaries. Thus a user who inadvertently mis-matches actual and formal parameters or who uses inconsistent common declarations is informed of his error at the time of linkage editing.

2.2 Statements

A IVTRAN source program consists of a set of statements each of which performs one of two functions:

-- it causes operations to be performed (e.g., arithmetic operations, input/output, or branching) or

-- it specifies program or data characteristics (e.g., array size or number and type of subprogram arguments).

The first type of statement is called an executable statement; the second is called a specification statement.

The statements of a program unit must be written in the following order:

-- Subprogram statement (BLOCK DATA, FUNCTION, or SUBROUTINE), if any.
-- IMPLICIT statement, if any.
-- other specification statements, if any.
-- Executable statements, at least one of which must be present, except in a block data subprogram, where executable statements are not allowed.
-- Debug statements, if any.
-- END statement.

The FORMAT, NAMELIST, and DATA statements may appear anywhere after the IMPLICIT statement, if present, and before the END statement.

Statement Layout

A program unit occupies one or more lines which are in turn divided into the following fields:

-- Label field - character positions 1 to 5. This field contains the attached label (statement number) or is blank. The label, if present, occupies one to five adjacent columns and is preceded or followed by one or more blanks. Blanks may not be imbedded within the label.
-- Continuation field - character position 6. This field is blank or zero if the line begins a statement and contains some other character if the line is the continuation of a statement begun on a previous line.
-- Statement field - character positions 7 to 72. This field contains the body of the statement.

-- Identification field - character positions greater than 72. This field is ignored by the compiler and may be used for whatever purpose the programmer desires. Traditionally, this field is used for identification and sequencing.

Comments are an exception to the above field usage, beginning with C,$, or * in the first character position and having any characters whatever in the remainder of the line. Comments are ignored by the compiler and are used to improve the readability of the program. A comment may not be continued on a subsequent line through use of the continuation field.

Statements are separated by the character $ within a line or by the end of the line followed by a comment line or a line whose continuation field is blank or zero.

Examples:

Comment:

```
        C     THIS IS A COMMENT
```

Multi-line statement:

```
      109     A = B +
        1     C * D
```

Multi-statement line:

```
              FUNCTION TWICE(X) $ TWICE = 2.0 * X $ RETURN $ END
```

Combination of multi-statement line and multi-line statement:

```
       25     IF(B .LT. C) GO TO 190 $ BIG = C $ K =
        X     K/ 2 $ L = L + 1
```

2.3 Expressions

The expression is used to compute a value to be stored in a variable, to be output, or to direct flow of control. Expressions are formed by combining operators (e.g., +,-,*,/) and operands (e.g., variables, constants, and function references). An expression may be either an array expression or a scalar expression. A scalar expression computes a single value. An array expression computes a set of values.

2.4 Tokens

Tokens are strings of characters which represent the objects and actions described in the language. There are the following types of tokens in the IVTRAN language:

- -- constant (e.g., 6.023)
- -- identifier. An identifier is a letter or a letter followed by one to five alphanumeric characters. An alphanumeric character is a letter or digit. (e.g., DELTA)
- -- label. A label is one to five decimal digits. (e.g., 2300)
- -- operator (e.g., .OR., .AND., +, -)
- -- separator (e.g., comma, parentheses, colon)
- -- alternate return (e.g., &12)
- -- keyword (e.g., DIMENSION, RETURN, INTEGER)

Unlike the blank in most other FORTRAN languages, the blank character in IVTRAN is significant. That is, blanks may not appear within a given token nor may a token be continued upon a second line (with the exception of the Hollerith constant). Blanks and continuations <u>may</u> be placed between any two tokens for purposes of readability. A blank or continuation <u>must</u> appear between two tokens if the first ends with a letter or digit and the second begins with a letter or digit. This convention is used in writing English sentences to prevent ambiguity and improve readability. For example, AN ICE HOUSE is different from A NICE HOUSE and neither is correctly written ANICEHOUSE.

To facilitate conversion of existing programs, the compiler is prepared to accept a single keyword in place of two adjacent keywords. For example, GOTO and GO TO will be considered equivalent as will BLOCKDATA and BLOCK DATA.

The formal syntax (Appendix A) does not take into consideration the effect of blanks. That is, it defines tokens and the legal combinations of tokens in a program unit, but does not define the optional or required blanks between any two tokens.

2.5 Characters

The characters used to write a IVTRAN program are the printing characters of the seven bit ANSI character set which are found on the ASR 33 keyboard plus blank (space); that is, codes 040_8 through 137_8. In addition, when lines are input through the teletype, the following characters serve to delimit lines:

-- carriage return, line feed. This is the standard line terminator.
  The following, though acceptable, are not normally used.
-- line feed
-- form feed
-- vertical tab
-- altmode (escape)

2.6 Data Types

A variable or function may have one of the following seven data types:

-- integer
-- double integer
-- real
-- double precision
-- complex
-- double complex
-- logical

Constants may be any of the above types and may also be of the following three data types:

-- Hollerith
-- octal
-- hexadecimal

The properties of data of each of the data types is given in Table 1.

The data type of a constant is implied by the form in which it is written. The data type of a variable or function may be specified in one of three ways:

-- Predefined specification contained in the IVTRAN language
-- IMPLICIT statement
-- Explicit specification statements

An explicit specification statement overrides an IMPLICIT statement, which, in turn, overrides the predefined specification.

The predefined specification declares a variable or function to be of type integer if the first letter of its name is I,J,K,L,M, or N and type real otherwise.

The IMPLICIT statement allows the programmer to associate a data type with an initial letter in much the same way the predefined specification does. The appearance of an initial letter in an IMPLICIT statement overrides the predefined specification.

A type or FUNCTION statement may specify the type of an identifier for a variable or function. Both methods of association of a data type with an identifier are described in greater detail in section 6.2.

TABLE 1: DATA TYPE PROPERTIES

Data Type	Mathematical Significance	Max. Value*	Min. Value*	Digits of Precision	Space Required (bits)
integer	exact integer	$2^{24}-1$	$-(2^{24}-1)$	up to 7^+	32
double integer	exact integer	$2^{48}-1$	$-(2^{48}-1)$	up to 14^+	64
real	approximation to real no.	$\pm 2^{64}$	$\pm 2^{-65}$	7^+	32
double precision	approximation to real no.	$\pm 2^{16384}$	$\pm 2^{-16385}$	14^+	64
complex	approximation to a pair of real numbers	same as two real numbers			64
double complex	approximation to a pair of real numbers	same as two double precision numbers			128
logical	true or false	--	--	--	1
Hollerith	character string	--	--	--	8n, where n is the number of characters
octal	exact integer	$2^{48}-1$	$-(2^{48}-1)$	up to 14^+	32 or 64
hexadecimal	exact integer	$2^{48}-1$	$-(2^{48}-1)$	up to 14^+	32 or 64

*Note:

$2^{24} = 16{,}777{,}216$ $\qquad$ $2^{-65} \simeq 2.71\mathrm{x}10^{-20}$

$2^{48} = 281{,}474{,}976{,}710{,}655$ $\qquad$ $2^{16384} \simeq 2.35\mathrm{x}10^{4930}$

$2^{64} \simeq 9.22\mathrm{x}10^{18}$ $\qquad$ $2^{-16384} \simeq 8.85\mathrm{x}10^{-4932}$

3. CONSTANTS

The data type of a constant is determined by the way in which a constant is written, constants of different types having different forms. A constant of integer, double integer, real, or double precision type may be signed. A signed constant is a constant of integer, double integer, real or double precision data type preceded by a plus or minus character. An optionally-signed constant is an unsigned constant or a signed constant.

3.1 Integer Constant

An integer constant is a string of decimal digits whose value is between zero and 16,777,215.

Examples:

Valid integer constants:

    0
    4754170
    01810

Invalid integer constants:

1.	(contains decimal point)
1,234,567	(contains commas)
20555000	(too large)

3.2 Double Integer Constant

A double integer constant is a string of decimal digits whose value is between 16,777,216 and 281,474,976,710,655. For smaller values, the programmer may use the integer constant and conversion will be performed.

Examples:

Valid double integer constants:

    23456789
    100000000000000

Invalid double integer constants:

0	(too small)
1000000000000000	(too large)

3.3 Real Constant

A real constant consists of a basic real constant, a basic real constant followed by a real exponent, or an integer constant followed by a real exponent. A basic real constant is a string of digits preceded by, containing, or followed by a decimal point. A real exponent is the character E followed by an optionally-signed integer constant. The value of a real constant is the value of the basic real constant or integer constant interpreted as a decimal fraction times ten to the value of the real exponent, if present. A real constant may assume values from approximately 2.71E-20 to 9.22E18 and zero.

Examples:

Valid real constants:

0.	
23.32	
34.56E3	(=34560.)
34.56E-3	(=.03456)
.45E+15	(=450000000000000.)
32E1	(=320.)

Invalid real constants:

99	(no decimal point)
6.023E23	(too big)
9.1066E-28	(too small)

3.4 Double Precision Constant

A double precision constant consists of a basic real constant followed by a double precision exponent or an integer constant followed by a double precision exponent. A double precision exponent is the character D followed by an optionally-signed integer constant. The value of a double precision constant is the value of the basic real constant or integer constant interpreted as a decimal fraction times ten to the value of the double precision exponent. A double precision constant may assume values from approximately 2.35D4930 to 8.85D-4932 and zero.

Examples:

Valid double precision constants:

6.023D23

9.1066D-28

300D+4900

Invalid double precision constants:

2.3D (missing exponent)

41.1D5000 (too big)

3.5 Complex Constant

A complex constant consists of a pair of optionally-signed real constants representing, respectively, the real and imaginary parts of a complex number, separated by a comma and enclosed in parentheses.

Examples:

Valid complex constants:

(1.,-2.) (=1-2i, where $i=\sqrt{-1}$)

(-.7071,-.7071)

(+34.5E10,-.22E-2)

Invalid complex constants:

(2. 3.) (missing comma)

(1,1) (neither real nor imaginary part a real constant)

(2.1E4,4.2D7) (imaginary part not a real constant)

3.6 Double Complex Constant

A double complex constant consists of a pair of optionally-signed double precision constants representing, respectively, the real and imaginary parts of a complex number, separated by a comma and enclosed in parentheses.

Examples:

Valid double complex constants:

(1D0,2D0)

(-.0001D4,+1.000D-4)

(4D4000,5D-4000)

Invalid double complex constants:

(3.,2.D30)	(real part not a double precision constant)
(.001D-1,.001E-1)	(imaginary part not a double precision constant)

3.7 Logical Constant

There are both logical scalar constants and logical array constants. The former represents a single logical value. The latter represents an array of logical values.

3.7.1 Logical Scalar Constant

A logical scalar constant is either the string of characters .TRUE. or the string of characters .FALSE.. The abbreviations, .T. and .F. are also allowed.

Examples:

Valid logical scalar constants:

.TRUE.	.T.	(both represent the value true)
.FALSE.	.F.	(both represent the value false)

Invalid logical scalar constants:

.TRUE	(decimal point is missing)
F	(both decimal points are missing)

3.7.2 Logical Array Constant

There are two forms of the logical array constant. Both forms specify which of the array elements are to assume the value true. All elements of the array which are not specified assume the value false. In either form the extent of the constant may either be implied by the basic logical array constant or explicitly attached.

3.7.2.1 Enumerated Logical Array Constant

An n-dimensional enumerated logical array constant is written as a list of n-dimensional index values, separated by commas and enclosed in brackets. An n-dimensional index value is written as exactly n integer constants, separated by commas and enclosed in parentheses. A one-dimensional index value may be optionally written as integer constant without enclosing parentheses. The extent is determined from the maximum value of the integer constants appearing in each index position.

Examples:

Valid enumerated logical array constants:

constant	extent	true value	false value
[(1,2,3),(2,2,2)]	(2,2,3)	(1,2,3),(2,2,2)	(1,1,1),(1,1,2),(1,1,3) (1,2,1), (1,2,2) (2,1,1),(2,1,2),(2,1,3) (2,2,1), (2,2,3)
[(2)]	(2)	(2)	(1)
[1,3,5,7]	(7)	(1),(3),(5),(7)	(2),(4),(6)
[ ] (4,6)	(4,6)	none	all
[1,3](6)	(6)	(1),(3)	(2),(4),(5),(6)

Invalid enumerated logical array constants:

[ ]	(Extent cannot be determined.)
[(2,1),(4)]	(Conflicting dimensionality for index values)
[(1,1),(2,2),(3,3)](4)	(Explicit extent does not agree in dimensionality with index values)
[(3,2,1),(1,2,3)](2,2,2)	(Index values exceed the extent)

3.7.2.2 Iterated Logical Array Constant

An iterated logical array constant specifies a one dimensional array (vector) of logical values. It is written in one of the forms:

[i,s...f]

or

[i...f]

where i,s, and f are integer constants representing the initial, second, and final index values for the true elements. If s is not specified, it is assumed to be i+1. The true values are all those elements j for which

$$i \leq j \leq f$$

and

$$j = i + n(s - i)$$

where n is a non-negative integer. The initial, second, and final values must be in increasing order of magnitude. That is,

$$0 < i < s < f.$$

Valid iterated array logical constants:

constant	extent	true values	false values
[1...5]	(5)	(1),(2),(3),(4),(5)	None
[1...5](10)	(10)	(1),(2),(3),(4),(5)	(6),(7),(8),(9),(10)
[2,4...10]	(10)	(2),(4),(6),(8),(10)	(1),(3),(5),(7),(9)
[1,4...9]	(9)	(1),(4),(7)	(2),(3),(5),(6),(8),(9)

Invalid iterated array logical constants:

[10...1]	(Final value less than initial value)
[-10...0]	(Initial and final values must be greater than zero)
[1...50](10)	(Final value larger than explicit extent)
[1,3,...,44]	(Extra commas)
[(1,1),(2,2)...(10,10)]	(Only one-dimensional arrays can be specified with this type of constant)

3.8 Hollerith Constant

There are two forms of the Hollerith constant. Each represents a character string of one or more printing characters and blanks. Hollerith constants are permitted only as initial values in the DATA statement and as arguments of a subroutine call or function reference.

3.8.1 Count-delimited Hollerith Constant

The count-delimited Hollerith constant is written as an integer constant with value n≤255 followed by the character H followed by exactly n characters which constitute the value of the constant. Blanks are legal within the n character string and are included in the count, n. End of line is permitted within the n-character string and is not included in the count n. This allows Hollerith constants to be created which have more than 69 characters.

Examples:

Valid count-delimited Hollerith constants:

2HIV

7H$199.95

14HFERDINAND FOCH

5HWON'T

Invalid coun -delimited Hollerith constants:

2HELP	(count too small)
5HHELP	(count too large, unless trailing blank is included)
10 HEXECUTIVES	(blank not permitted between n and H)

3.8.2 Quote-delimited Hollerith Constant

The quote-delimited Hollerith constant is written as string of n≤255 characters enclosed in apostrophes.* Within this string, an apostrophe is represented as a pair of adjacent apostrophes. End of line and blank are permitted within the string. Blank is a part of the string. End of line is not.

Examples:

Quote-delimited	Count-delimited equivalent
'TEXT'	4HTEXT
'DON''T'	5HDON'T
'FO''C''S''LE'	9HFO'C'S'LE
'$ IS A DOLLAR SIGN'	18H$ IS A DOLLAR SIGN

3.9 Hexadecimal Constant

The hexadecimal constant represents an integer or double integer value as a number with radix 16. The digits with values 10 through 15 are represented by the letters A through F. A hexadecimal constant is written as a string of hexadecimal digits immediately preceded by the letter Z. If the value of the string, interpreted as a number in the hexadecimal number system, is less than 2^{24} the constant represents a value of integer data type. If the value is greater than or equal 2^{24} but less than 2^{48} the constant represents a value of double integer type. In no case may the value

* Called "single quote" by some programmers.

exceed or equal 2^{48}. The hexadecimal constant is permitted only as an initial value within the data statement.

Valid hexadecimal constants:

ZA	$(=10_{10})$
ZFF	$(=255_{10})$
ZFACECAB054CA	(=a double integer)

Invalid hexadecimal constants:

ZBRA	(R is not a valid hexadecimal digit)
ZAAABBBCCCDDDE	(too large)
Z 12	(blank not permitted)

3.10 Octal Constant

The octal constant represents an integer or double integer value as a number with radix 8. An octal constant is written as a string of octal digits either preceded by the letter O or followed by the letter B. If the value of the string, interpreted as an octal number, is less than 2^{24} the constant represents a value of the integer data type. If the value is greater than or equal to 2^{24} and less than 2^{48} the constant represents a value of the double integer data type. In no case may the value exceed 2^{48}. The octal constant is permitted only as an initial value in the DATA statement.

Examples:

Valid octal constants:

O77	77B	(both $= 63_{10}$)
O77777777		$(= 2^{24}-1)$
100000000B		$(= 2^{24}$, a double integer value)

Invalid octal constants:

O377600B	(either the O or the B but not both are permitted)
O10000000000000000	(too big)
789B	(8 and 9 are not octal digits)

4. VARIABLES

A variable is a quantity whose value may change during the execution of a program. A variable is associated with a storage area within a program. Both scalar and array variables have an associated data type which is determined through the use of the type statement, the IMPLICIT statement, or through the predefined type. Array variables have as associated extent which must be declared in a specification statement and an associated allocation which is either defined in a specification statement or is the predefined, default allocation. Throughout the body of this manual, default allocation is assumed. Other allocations are described in Appendix B.

4.1 Scalar Variable

The scalar variable represents a single quantity and is associated with a single storage location. A scalar variable is referred to by an identifier.

Examples:

Valid scalar variable names:

```
A
PITCH
VAT69
ICURYY
```

Invalid scalar variable names:

99THMP	(does not begin with a letter)
PRESSURE	(over six characters)
CA$H	($ not a letter or a digit)

4.2 Array Variable

An array variable represents a collection of values of a single data type and is associated with a set of storage locations. The number and structure of the values is determined by the array extent. The arrangement in storage is determined by the array allocation. A complete array is referred to by an identifier. A single array element is referred to by writing the array name followed by a subscript. Subarrays of the array are referred to by an array cross-section which fixes one or more subscripts while allowing the remainder to vary.

4.2.1 Array Extent

An n-dimensional array has an extent which is written as a list of n non-zero integer constants, each called a dimension, separated by commas and enclosed in parentheses. The number of elements in the array is the product of the dimensions. Each dimension determines the maximum value of the corresponding subscript position in array element references and array cross-section references. An array which is a dummy parameter to a function or subroutine may have one or more variable dimensions. Each variable dimension is an integer scalar variable passed to the function or subroutine as a parameter or in a common block. A variable dimension may not be modified within the function or subroutine and must have the same value as the corresponding dimension of the actual argument.

Example:

Extent	Legal subscript values
(5)	(1),(2),(3),(4),(5)
(3,4)	(1,1),(2,1),(3,1)
	(1,2),(2,2),(3,2)
	(1,3),(2,3),(3,3)
	(1,4),(2,4),(3,4)
(3,2,3)	(1,1,1),(2,1,1),(3,1,1)
	(1,2,1),(2,2,1),(3,2,1)
	(1,1,2),(2,1,2),(3,1,2)
	(1,2,2),(2,2,2),(3,2,2)
	(1,1,3),(2,1,3),(3,1,3)
	(1,2,3),(2,2,3),(3,2,3)

4.2.2 Array Element

An array element is referred to by writing an n-dimensional array name followed by an n-dimensional subscript. An n-dimensional subscript is a parenthesized list of n scalar expressions of type integer separated by commas. Double integer, real, and double precision scalar expressions are also permitted and will be converted to integer type before accessing the array element. At the time the array element reference is executed the scalar expressions must each evaluate to an integer between 1 and the corresponding dimension of the array. The result is a scalar value of the same data type as that of the array name.

Examples:

Valid array element references:

```
PRIDE(LIONS)
GRID (I+1;J-3)
GROSS (12,12)
```

Invalid array element references:

Array	Extent	Reference	
DOZEN	(12)	DOZEN (-3)	(subscript value must be between 1 and 12)
SPACE	(3,3,3)	SPACE(I,J)	(dimensionality of subscript must equal dimensionality of array)

4.2.3 Array Cross-Section

An array cross-section is referred to by writing the array name followed by a parenthesized list of subscript expressions and asterisks separated by commas. Each subscript expression is a scalar integer expression corresponding to a fixed subscript. Each asterisk corresponds to a subscript which varies over its allowable range. The subscript expressions may be of double integer, real, or double precision data types and will be converted to integer type before accessing the array cross-section. The result is an array of the same data type as the array cross-section name with an n-dimensional extent where n is the number of asterisks in the reference. The ith dimension of the resultant array is the same as the j_1th dimension of the base array, where j_i is the index of the ith asterisk.

Examples:

Extent	Cross-section	Result Extent
(5,60)	B(I,*)	(60)
	B(*,I)	(5)
(300,20,4)	A(I,J,*)	(4)
	A(I,*,K)	(20)
	A(I,*,*)	(20,4)
	A(*,J,K)	(300)
	A(*,J,*)	(300,4)
	A(*,*,K)	(300,20)

5. EXPRESSIONS

An expression computes a value or set of values for use within a statement. A scalar expression computes a single value. An array expression computes a set of values. A scalar expression has of one of the declarable data types: integer, double integer, real, double precision, complex, double complex, or logical. An array expression has one of the declarable data types and an extent.

5.1 Expression Form

An expression is composed of operators, which specify operations to be performed, and primaries (operands), which specify the values upon which the operations are to be performed.

5.1.1 Primaries

A primary is a constant (chapter 3), a variable, array element, or cross-section (chapter 4), a function reference (section 7.3), a set selector (section 5.1.1.1), or a parenthesized expression.

Examples:

(-3.2,4E-6)	complex constant
(3*I-14)	parenthesized expression
SIN(X)	function reference
A	variable
A(I-1,J+1)	array element
A(*,K,L-1)	cross-section
[(I)/[1...50]:I.LT.J]	set selector

5.1.1.1 Set Selector

A set selector is written:

[(I)/ S:B]

where I is an integer scalar variable, S is 1-dimensional logical array expression and B is a logical scalar expression which may depend upon I.

Within B all array element references must be either independent of I or have exactly one subscript of the form:

I

or

I + C

or

I - C

where C is an expression independent of I.

The result of evaluating a set selector is a one-dimensional logical array with the same extent as S and whose values (J) are true if S(J) is true and B is true when evaluated with I=J.

Examples:

Valid set selectors:

Set selector	Result Value
[(I)/[1...10]:I.GT.7]	[8,9,10](10)
[(I)/[1...100]:I**3-6*I**2+11*I-6.EQ.0]	[1,2,3](100)
[(I) /[1...50]:I.LT.14.AND. I.GT.7]	[8,9,10,11,12,13](50)
[(L)/[1...99]:L .EQ.L+1]	[ ](99)

Invalid set selectors:

[(I,J)/[(1,1),(2,2)]:I+J.EQ.2]	(Invalid with default allocation. See Appendix B.)
[(I)/[1,1),(2,2)]:I.GT.1]	(logical array expression must be singly dimensional)
[(I)/[1...100]: A(2*I).EQ.1.0]	(improper use of I in array reference)

5.1.2 Complete Expressions

An expression is either a primary, a unary operator followed by an expression, or an expression followed by a binary operator followed by an expression. The order of operations is determined by the precedence between operators, the operator with the greater precedence being evaluated first. If the precedence of operators is the same,non-cummutative operations are performed in left-to-right order and commutative operators may be performed in arbitrary order. A unary operator may only follow an operator of lower precedence. The legal operators, operands, and precedence are given in Table 2.

TABLE 2: OPERATORS

Precedence	Operator	Meaning	Operand Types[1]	Result Type
Arithmetic Operators:				
8 (highest)	**(binary)	exponentiation[3]	arithmetic	arithmetic[2]
7	*(binary)	multiplication	arithmetic	arithmetic[2]
	/(binary)	division[3]	arithmetic	arithmetic[2]
6	+ (unary)	positive sign[3]	arithmetic	arithmetic[2]
	- (unary)	negation[3]	arithmetic	arithmetic[2]
	+ (binary)	addition	arithmetic	arithmetic[2]
	- (binary)	subtraction[3]	arithmetic	arithmetic[2]
Relational Operators (binary):				
5	.LT.	less than[3]	arithmetic except complex	logical
	.LE.	less than or equal to[3]	arithmetic except complex	logical
	.EQ.	equal	arithmetic	logical
	.NE.	not equal	arithmetic	logical
	.GT.	greater than[3]	arithmetic except complex	logical
	.GE.	greater than or equal[3]	arithmetic except complex	logical
Logical Operators:				
4	.CROSS.,.C. (binary)	cross product[3]	logical array	logical array
3	.NOT.,.N. (unary)	logical negation[3]	logical	logical
2	.AND.,.A.	logical product	logical	logical
	.DIFF.,.D. (both binary)	logical difference[3]	logical	logical
1 (lowest)	.OR.,.O.	logical sum	logical	logical
	.XOR.,.X. (both binary)	logical exclusive or	logical	logical

Notes:

1) Arithmetic means integer, double integer, real, double precision, complex, or double complex. Arithmetic except complex means integer. double integer, real, or double precision.
2) See Table 3.
3) Non-commutative operator.

Examples:

Valid expressions:		Equivalent expressions:
A+B		
SIN(X)/ COS(X+1.0)		
A*B+C		(A*B)+C
A+B*C		A+B*C
A/ B/ C		(A/ B)/ C
A*B*C		(A*B)*C
	or	A*(B*C)
	or	(A*C)*B
A.LT.B+C.OR.TEST.AND.GUESS		(A.LT.(B+C)).OR.TEST.AND.GUESS)
FLAG.AND.NOT.-A+B .LT.-A*B		FLAG.AND.(.NOT.(((-A)+B).LT.(-(A*B))))

Invalid expressions:	
A*-B	(unary operator follows operator with higher precedence)
A.LT.B.GT.C	(.LT. gives logical result but .GT. requires arithmetic operands)
A+B .OR. C/D	(+ and / give arithmetic results but .OR. requires logical operands)
A .NOT. .LE. B	(use .NOT. A .LE. B or A .GT. B instead)

5.2 Expression Type

The type of a parenthesized expression is that of its operand. The type of a relational or logical expression is logical. The type of an arithmetic expression is determined by the type of the operand(s). The type of an arithmetic expression formed with a unary operator is that of the operand. The type of an arithmetic expression formed with a binary operator is given in Table 3.

TABLE 3: ARITHMETIC OPERATOR RESULT TYPES

	Operand 2:					
Operand 1:	Integer	Double Integer	Real	Double Precision	Complex	Double Complex
Integer	Integer	Double Integer	Real	Double Precision	Complex	Double Complex
Double Integer	Double Integer	Double Integer	Double Precision	Double Precision	Double Complex	Double Complex
Real	Real	Double Precision	Real	Double Precision	Complex	Double Complex
Double Precision	Double Precision	Double Precision	Double Precision	Double Precision	Double Complex	Double Complex
Complex	Complex	Double Complex	Complex	Double Complex	Complex	Double Complex
Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

Examples:

Let I be integer, DI be double integer, R be real, DP be double precision, C be complex, and DC be double complex.

Expression	Result Type	
I+DI	DI	
C+DI	DC	
R**R	R	
I*R+DI	DP	
DI+R+C	DC	(note that the result type is independent of the order of execution of commutative operators)
I/ I	I	

5.3 Expression Extent

The extent of a parenthesized expression is that of the operand. The extent of an expression formed with a unary operator is that of the operand. The extent of an expression formed with a binary operator is given in Table 4 for all binary operators except .CROSS. For that operator, the extent is the concatenation of the extents of the two operands.

Example:

A has extent (3,40) and B has extent (500,6000).
A.CROSS.B has extent (3,40,500,6000).
B.CROSS.A has extent (500,6000,3,40).

TABLE 4: Expression Extent for Binary Operators

Operand 1	Operand 2	Result Extent
Scalar	Scalar	Scalar.
Scalar	Array	Same as that of operand 2. Operand 1 is applied to each element of Operand 2.
Array	Scalar	Samo as that of operand 1. Each element of operand 1 is applied to Operand 2.
Array	Array	Same as that of operands 1 and 2, which must have identical extents.

Example:

Let A, B and C be real arrays with extent (3) and the following values:

A (1) = 0.0
A (2) = 1.0
A (3) = 2.0

B(1) = .25
B(2) = .5
B(3) = 1.0

C = A/B yields

C(1) = 0.0
C(2) = 2.0
C(3) = 2.0

C = (A+B)*4.0 yields

C(1) = 1.0
C(2) = 6.0
C(3) = 12.0

C = 2. -A yields

C(1) = 2.0
C(2) = 1.0
C(3) = 0.0

5.4 Operators

Each operator listed in Table 2 has a different interpretation and set of legal values.

5.4.1 Arithmetic Operators

Each of the arithmetic operators has the usual mathematical interpretation. When operands of dissimilar data types are combined, the operands are first converted to the result data type and then the operation is performed.

Integer and double integer division produce a result which is truncated to the nearest integer whose magnitude does not exceed the magnitude of the mathematical value represented by the division.

Division by zero is not defined.

A negative base may not be raised to a non-integral power unless either the base or the power or both is of complex or double complex data type. A zero valued base may not be raised by a zero valued exponent.

No result may be evaluated which yields a value outside the range of values permitted for the result data type.

Examples:

Valid arithmetic operations:	Result:
3+4	7
2.5 * 4.0	10.0
1/3	0
1./3	.33333333
1/3.	.33333333
1./3.	.33333333
15/7	2
(1.1,-2.2)-(.9,4.0)*2	(-.7,-10.2)
Invalid arithmetic operations:	
2/0	(division by zero not permitted)
(-3.)**(.5)	(negative base may not be raised to non-integral negative power unless complex)
0 ** 0	(zero base may not be raised to zero exponent)
2**100	(value out of range)
1E18*2E18	(value out of range)

5.4.2 Relational Operators

The relational operators perform comparisons between arithmetic values. The result is true if the relation is satisfied and false otherwise. When types are dissimilar, the comparison is performed after conversion to the result type shown in Table 2.

Examples:

Valid relational operations:	Result:
3.5 .LT. 4.5	true
1+1 .EQ. 2	true
.1+.1 .EQ. .2	(may not be true due to round-off and truncation error)
1/3 .GE. 1./3.	false (1/3 = 0)
(1.,0.) .EQ. 2	false
Invalid relational operations:	
(1.,0.) .LE. 2	(complex only valid for .EQ. and .NE.)

5.4.3 Logical Operators

The result of each of the logical operators except .CROSS. is given in Table 5. The result of A.CROSS.B is an array C whose elements are defined by the following equation:

$$C(I_1,I_2,\ldots,I_n,J_1,J_2,\ldots,J_m) = A(I_1,I_2,\ldots,I_n).AND.B(J_1,J_2,\ldots,J_m).$$

Example:

Let A = [1,3,4].
Let B = [2,3,5].

```
A.CROSS.B = [ (1,2),(1,3),(1,5),
              (3,2),(3,3),(3,5),
              (4,2),(4,3),(4,5)]
```

TABLE 5: Results of Logical Operators

Operand 1: Operand 2:	false false	false true	true false	true true
Operator:				
.NOT.,.N.	true	---	---	false
.AND.,.A.	false	false	false	true
.DIFF.,.D.	false	false	true	false
.OR.,.O.	false	true	true	true
.XOR.,.X.	false	true	true	false

Examples:

Let Y = .TRUE. and N = .FALSE. .

Valid logical operations:	Result:
Y .AND. N	false
Y .OR. N	true
N .XOR. Y	true
Y .AND. .NOT. N	true
[1...100] .AND. N	[ ] (100)
[2,4...20] .AND. [3,6...20]	[6,12,18]
[2,4...20] .DIFF. [3,6...20]	[2,4,8,10,14,16,20]
.NOT. [1,3...10]	[2,4...10]
[1,2,3,5,8,13](100) .AND.[1,3...100]	[1,3,5,13](100)
[2,3].C.[3,4].XOR.[1,2].C.[3]	[(1,3),(2,4),(3,3),(3,4)]
[ ](10).C.[ ] (20)	[ ](10,20)

Invalid logical operations:	
[1,2,3,5,8,13].AND.[1,3...100]	(extents of operands not equal. Use [1,2,3,5,8,13](100).)
Y.NOT. .AND. N	(use .NOT. Y .AND. N, Y .AND. .NOT. N or .NOT. (Y .AND. N) instead)

6. STATEMENTS

A statement is an executable statement or a specification statement. Executable statements specify actions; specification statements describe the characteristics of and arrangement of data, editing information, statement functions, and the characteristics of and classification of program units.

6.1 Executable Statements

There are four types of executable statements:

1) Assignment statements
2) Control statements
3) Input/Output statements
4) Debug statements

6.1.1 Assignment Statements

There are three types of assignment statements:

1) Arithmetic assignment statement
2) Logical assignment statement
3) GO TO assignment statement

6.1.1.1 Arithmetic Assignment Statement

An arithmetic assignment statement is one of the forms:

v = s

or a = s

or a = e

where v is an arithmetic scalar variable or array element reference, s is an arithmetic scalar expression, a is an arithmetic array or array cross-section reference, and e is an arithmetic array expression. Execution of the statement causes the expression s or e to be evaluated, converted to the data type of v or a according to Table 6, and assigned to v or a. The extent of a must be the same as the extent of e. If the second form is used, the value of s is assigned to each element of a.

Examples:

Let A be a real array with extent (5,6),
B be a real array with extent (5),
I be a scalar integer, and
J and K be double integer scalars.

Valid arithmetic assignment statements:

J = K*2 — assigns the value of the expression K*2 to J.

I = J — converts the value of J to integer type and assigns it to I.

A = 4.5 — assigns the value 4.5 to each of the elements of A.

A(*,I)=B — assigns each element of B to the corresponding element of the cross-section of A.

B(I) = J+A(I,3)-J**K — converts the value of the expression to real type and assigns it to the array element B(I).

Invalid arithmetic assignment statements:

A = B — (incompatible extents)

K = .TRUE. — (incompatible data types)

6.1.1.2 Logical Assignment Statement

A logical assignment statement is one of the forms:

v = s
or a = s
or a = e

where v is a logical scalar variable or array element reference, s is a logical scalar expression, a is a logical array or array cross-section, and e is a logical array expression. Execution of the statement causes the expression s or e to be evaluated and assigned to v or a. The extent of a must be the same as the extent of e. If the second form is used, the value of s is assigned to each element of a.

TABLE 6: DATA TYPE CONVERSION

	To:					
From:	Integer	Double Integer	Real	Double Precision	Complex	Double Complex
Integer	--	convert	convert	convert	convert	convert
Double Integer	convert*	--	truncate	convert	truncate	convert
Real	fix*	fix*	--	convert	convert	convert
Double Precision	fix*	fix*	truncate*	--	truncate*	convert
Complex	fix real part*	fix real part*	take real part	convert real part	--	convert
Double Complex	fix real part*	fix real part*	truncate real part*	take real part	truncate*	--

Notes:

1) Convert means to change the form of the datum without changing its value.
2) Truncate means that only the most significant digits will be retained in the process of conversion.
3) Fix means that only the integer part of the value is retained.

*Operations marked with an asterisk can result in overflow; that is, values can result which are outside of the allowable range for the type to which the data is converted.

Examples:

Let L be a logical array with extent (9)
S be a logical array with extent (9,2), and
T be a logical scalar.

Valid logical assignment statements:

```
L = T
S = [1,4,9].CROSS.[2]
S(*,1) = L.DIFF.[4](9)
T = .TRUE.
S(J,*) = .FALSE.
```

Invalid logical assignment statements:

```
S = [1,4,3]              (incompatible extent)
L = [1,4]                (incompatible extent; use
                         [1,4](9) instead)
L = 3.5                  (incompatible data type)
```

6.1.1.3 GO TO Assignment Statement

A GO TO assignment statement is of the form:

```
ASSIGN k TO i
```

where k is a statement label and i is an integer scalar variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement using that integer scalar variable will cause the statement identified by the assigned statement label k to be executed next, provided there has been no intervening reassignment of the variable. The statement label must refer to an executable statement in the same program unit in which the ASSIGN statement appears. Once having been mentioned in an ASSIGN statement, as integer scalar variable may not be referenced in any statement other than an assigned GO TO statement until it has been reassigned a numeric value.

Example:

```
ASSIGN 32 TO KZ                At this point the label 32 is assigned
                               to the Integer scalar variable named KZ.
   .
   .
   .
GO TO KZ,(451,7,32,110)        At the execution of this statement control
                               is transferred to the statement labelled 32.
```

6.1.2 Control Statements

There are seven types of control statements:

1) GO TO statements
2) IF statements
3) CALL statement
4) RETURN statement
5) CONTINUE statement
6) Program control statements
7) Loop statements

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

6.1.2.1 GO TO Statements

There are three types of GO TO statement:

1) Unconditional GO TO statement
2) Assigned GO TO statement
3) Computed GO TO statement

6.1.2.1.1 Unconditional GO TO Statement

An unconditional GO TO statement is of the form:

GO TO k

where k is a statement label. Execution of this statement causes the statement identified by the statement label k to be executed next.

Example:

```
GO TO  1066
```

6.1.2.1.2 Assigned GO TO Statement

An assigned GO TO statement is of the form:

GO TO i, $(k_1, k_2, \ldots, k_n)$

where i is an integer scalar variable reference, and the k's are statement labels. The use of the comma after the integer scalar variable reference is optional. The order in which the k's are written is optional.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement (6.1.1.3) to be one of the statement labels in the parenthesized list, and such an execution causes the statement identified by that statement label to be executed next.

Example:

```
GO TO CHOICE, (123,5813,2235,5792,141)
```

6.1.2.1.3 Computed GO TO Statement

A computed GO TO statement is of the form:

GO TO$(k_1,k_2,\ldots,k_n)$,i

where the k's are statement labels and i is an integer scalar variable reference. The use of the comma before the integer scalar variable reference is optional. Execution of this statement causes the statement identified by the statement label k_j to be evaluated next, where j is the value of i at the time of execution. This statement is defined only for values of j such that $1 \le j \le n$.

Example:

```
I = 3
GO TO (3,17,45,4,9),I
```

Control is transferred to the statement labeled 45.

6.1.2.2 IF Statements

There are three types of IF statement:

1) Arithmetic IF statement
2) Logical IF statement
3) Two-branch IF statement

6.1.2.2.1 Arithmetic IF Statement

An arithmetic IF statement is of the form:

IF(e)k_1,k_2,k_3

where e is a scalar expression of typo integer, double integer, real, or double precision and the k's are statement labels. The arithmetic IF is a three-way branch. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label k_1,k_2, .or k_3 is executed next as the value of e is less than zero, zero, or greater than zero, respectively.

Examples:

	Control transferred to:

VALUE = 3.7
IF(VALUE)96,1,42
	42

VALUE = -43E+17
IF(VALUE)96,1,42
	96

VALUE = 0.0
IF(VALUE)96,1,42
	1

6.1.2.2.2 Logical IF Statement

A logical IF statement is of the form:

IF(e)S

where e is a logical expression and S is any executable statement except a DO, DO FOR ALL, or another logical IF statement. Upon execution of this statement the logical expression e is evaluated. If the value of e is false, statement S is executed as though it were a CONTINUE statement. If the value of e is true, statement S is executed.

Examples:

Valid logical IF statements:

```
IF(A .LE. 0.0) GO TO 17
IF(FROST) ALPINE = TUNDRA
IF(SIN(X) .GT. .5) Y(I) = COS(X)
IF(P .OR. Q) IF (L+M) 4,2,7
```

<u>Invalid</u> logical IF statements:

```
IF(4+I) GO TO 405                     (4+I is not a logical expression)
IF(A .LT. B) IF (P .NE. 2.) X=A+B     (object of IF must not be another
                                      logical IF)
```

6.1.2.2.3 Two-branch IF Statement

A two-branch IF statementis of the form:

$\mathrm{IF}(e)k_1,k_2$

where e is a logical scalar expression and k_1 and k_2 are statement labels. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label k_1 or k_2 is executed next as the value of e is true or false, respectively.

Example:

```
IF(A.LT.B) 43,80
```

6.1.2.3 CALL Statement

A CALL statement is one of the forms:

CALL $s(a_1,a_2,\ldots,a_n)$

or

CALL s

where s is the name of a subroutine or subroutine entry point and the a's are actual arguments (7.1.2). The inception of execution of a CALL statement references the designated subroutine or subroutine entry point. Such a reference causes execution to proceed with the first executable statement in the subroutine or the first executable statement after the corresponding ENTRY statement, respectively. Return of control from the designated subroutine completes execution of the CALL statement.

Examples:

```
CALL ME(ISH,MAEL)
CALL OUS(SIN(X)*P-COS(X)*Q)
CALL IOPE
```

6.1.2.4 RETURN Statement

A RETURN statement is one of the forms:

RETURN

or

RETURN i

where i is a scalar integer expression. The RETURN statement marks the logical end of a main program or a function or subroutine subprogram. The second form may only be used in a subroutine subprogram.

Execution of this statement when it appears in a function subprogram causes return of control to the current calling program unit. At this time the value of the function is returned as the value of the function reference.

Execution of this statement when it appears in a subroutine subprogram causes return of control to the current calling program unit. If the first form is used control is transferred to the first executable statement which follows the corresponding CALL statement. If the second form is used, the expression i is evaluated and control is returned to the jth alternate return, where j is the value of i. If the number of alternate returns is n then j must be greater than zero and less than or equal to n.

Execution of this statement when it appears in a main program is equivalent to the execution of a STOP statement.

6.1.2.5 CONTINUE Statement

A CONTINUE statement is of the form:

CONTINUE

Execution of this statement causes continuation of the normal execution sequence.

6.1.2.6 Program Control Statements

There are three program control statements:

1) STOP statement
2) PAUSE statement
3) END statement

6.1.2.6.1 STOP Statement

A STOP statement is one of the forms:

STOP n

or

STOP

Where n is an integer constant. Execution of this statement causes termination of the executable program. At that time n, if specified, is output.

6.1 2.6.2 PAUSE Statement

A PAUSE statement is one of the forms:

PAUSE

or

PAUSE n

or

PAUSE h

where n is an integer constant and h is a Hollerith constant. This statement is provided for compatability only and in IVTRAN has the same effect as a CONTINUE statement. In other processors, this statement would cause cessation of the program in such a way that resumption would be at the discretion of the operator.

6.1.2.6.3 END Statement

The END statement is one of the forms:

END

or

END s

where s is the name of the subprogram of which the END statement is a part. The END statement must be the last statement, physically, in any program unit. The complete END statement, including the subprogram name, must appear on a single line; it may not be continued onto a second line through the use of the continuation field.

Execution of the END statement is equivalent to the execution of a RETURN statement in a function or subroutine subprogram. It is equivalent to the execution of a STOP statement in a main program.

6.1.2.7 Loop Statements

There are two types of loop statements:

1) DO statement

2) DO FOR ALL statement

6.1.2.7.1 DO Statement

A DO statement is one of the forms:

DO n i = m_1, m_2, m_3

or

DO n i = m_1, m_2

where:

1) n is the statement label of an executable statement. This statement, called the terminal statement of the DO loop, must physically follow and be in the same program unit as the DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, two-branch IF, RETURN, STOP, or another DO statement.

2) i is a scalar integer variable name; this variable is called the control variable.

3) m_1, called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or an integer scalar variable name. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2, and m_3 must each be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first statement following the DO, to and including the terminal statement associated with the DO. In case the range includes another DO statement, the range of the contained DO must be a subset of the containing DO.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following six steps:

1) The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

2) The range of the DO is executed.

3) If contiol reaches the terminal statement, then after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

4) If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal paramenter, then the action described starting at step 2 is repeated, with the understanding that the range in question is that of the DO whose control variable has been most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, then the DO is said to have been satisifed and the control variable becomes undefined.

5) At this point, if there were one or more other DO statements referring to the terminal statement in question, the contiol variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which the first executable statement following the terminal statement is executed.

6) Upon exiting from the range of a DO by a transfer of control as opposed to satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the preceding paragraphs.

An alternative but equivalent definition of the execution of the DO statement is the following:

1) Replace each DO statement of the form:

```
        DO n i = $m_1$,$m_2$
```

by

```
        DO n i = $m_1$,$m_2$,$m_3$
```

where m_3 is the integer constant 1.

2) Replace, starting with the innermost DO, each DO loop of the form:

```
        DO n i = $m_1$,$m_2$,$m_3$
        range
n       terminal statement
```

by

```
        i = $m_1$
k       CONTINUE
        range, with all occurences of n replaced by n1
n1      terminal statement
        i = i + $m_3$
        IF(i .LE. $m_2$) GO TO k
n       i = ?
```

where n1 and k are labels not appearing elsewhere in the program unit and ? is an integer constant of unknown value.

Both of the above definitions permit transfer out of the range of the DO and subsequent re-entry by transfer of control to a statement within the range of the DO loop. The statements executed between the transfer out of the range of the loop and the subsequent re-entry are called the extended range of the DO. Control may not be transferred into the range of a DO except through the execution of the DO statement or through the use of an extended range.

The control variable, terminal parameter, and incrementation parameter may not be assigned values within the range or extended range of a DO loop.

Examples:

Valid DO loops:

1)
```
          DO 3 K = 1,9,2
      3   A(K) = B(K)*2.0
```

is equivalent to:

```
          K = 1
    k     CONTINUE
    n1    A(K) = B(K)*2.0
          K = K + 2
          IF (K .LE. 9) GO TO k
          K = ?
```

or to:

```
          A(1) = B(1)*2.0
          A(3) = B(3)*2.0
          A(5) = B(5)*2.0
          A(7) = B(7)*2.0
          A(9) = B(9)*2.0
```

2)
```
          NI = N-1
          DO 4 I = 1,N1
          I1 = I + 1
          DO 4 J = I1,N
          IF(A(I) .LE. A(J) ) GO TO 4
          T = A(I)
          A(I) = A(J)
          A(J) = T
      4   CONTINUE
```

is equivalent to:

```
        N1 = N - 1
        I = 1
   k1   CONTINUE
        I1 = I + 1
        J = I1
   k2   CONTINUE
        IF(A(I) .LE. A(J)) GO TO n1
        T = A(I)
        A(I) = A(J)
        A(J) = T
   n1   CONTINUE
        J = J + 1
        IF(J .LE. N) GO TO k2
   n2   J = ?
        I = I + 1
        IF (I .LE. N1) GO TO k1
    4   I = ?
```

Invalid DO loops:

1)

```
    3   CONTINUE
        A(I) = I
        DO 3 I = 1,10
```

(Terminal statement must follow DO)

2)

```
        DO 3 I = 1,N
        DO 4 J = 1,M
    3   A(I) = I
    4   B(J) = J
```

(Inner DO must be completely contained in outer DO.)

3)

```
        GO TO 7
        DO 4 I = 1,N
        A(I) = I
    7   B(I) = I-1
    4   CONTINUE
```

(Transfer into DO not permitted except as part of an extended range)

4)

```
        DO 1 I = 0,N
    1   A(I+1) = I
```

(DO parameters must be positive non-zero integers)

5)

```
      DO 12 J = 10,1
   12 A(J+1) = A(J)
```

(Initial parameter must not exceed final parameters)

6)

```
      DO 15 KP = N,M,2
      A(KP) = SIN (B(KP))
      KP = KP-1
   15 A(KP) = COS (B(KP)
```

(DO parameters and index may not be modified within loop)

6.1.2.7.2 DO FOR ALL Statement

A DO FOR ALL Statement is of the form:

DO k FOR ALL (i)/s

where:

1) k is the statement label of an executable statement. This statement, called the terminal statement of the associated DO FOR ALL, must physically follow and be in the same program unit as the DO FOR ALL statement.

2) i is an integer scalar variable name called a control index.

3) s is an one-dimensional logical array expression with an extent (d).

Associated with the DO FOR ALL is a range defined to be those executable statements following the DO FOR ALL from and including the first statement following the DO FOR ALL to and including the terminal statement associated with the DO FOR ALL.

It is not permitted to transfer control into the range of a DO FOR ALL loop except by executing a DO FOR ALL statement. It is not permitted to transfer out of the range of a DO FOR ALL loop except by executing the terminal statement which allows flow to proceed to the first executable statement following the loop.

A DO FOR ALL statement is used to specify that certain assignment statements within its range are to be executed for a set of values in parallel. This set of values, called the index set, is defined to be the set of integers i such that s(i) is true and $1 \leq i \leq d$.

The range of a DO FOR ALL may not contain another DO FOR ALL or any array expressions. The control indices may not be used in any statement within the range of the DO FOR ALL loop, except as outlined in the following paragraphs. Otherwise any statement permitted elsewhere in the procedure part of the program is permitted, and has its usual interpretation.

Within a DO FOR ALL range the control indices may appear only in DO FOR ALL assignment statements. A DO FOR ALL assignment statement is one of the following forms:

```
        p = e
or
        IF(f) p = e
```

where

1) p is an array element reference with exactly one subscript of the form:

```
        I
or      I + C
or      I - C
```

where I is the control index and C is an expression independent of I.

2) e and f are each an expression which may or may not depend upon I. Within e and f any array element references are either of the form 1) above or are independent of I.

Execution of a DO FOR ALL assignment statement causes parallel assignment of the expression e to the array element reference p for all values I in the index set. The computation of the expressions e and f makes use of values of p in effect immediately before execution of the statement. If the second form of the statement is used, assignment is made only for those values of I in the index set for which f(I) = true.

<u>Examples</u>:

<u>Valid</u> DO FOR ALL loops:

```
1)      DO 1 FOR ALL (I)/[2...99]
    1   A(I) = A(I-1)*A(I+1)

2)      DO 46 FOR ALL (K)/[1,3...100]
   46   IF (VAR(K) .LT. 0.0) VAR(K) = -(VAR(K)

3)      DO 77 FOR ALL (J)/ B(M,*,N)
        BIG(J) = 0.0
        DO 77 I = 1,M
   77   IF(A(I,J) .GT. BIG(J)) BIG(J) = A(I,J)
```

Invalid DO FOR ALL loops:

1)

```
        DO 64 FOR ALL (L)/[1...100]
        DO 64 FOR ALL (M)/[1...100]
     64 A(L,M) = B(L,M)
```

DO FOR ALL loops may not be nested.

2)

```
        DO 10 FOR ALL (IZ)/ LARR
     10 A(IZ,*) = B(*,IZ)
```

A DO FOR ALL loop may not contain array expressions.

3)

```
        DO 900 FOR ALL (I)/[1...40]
    900 A(2*I) = A(I+40)*I
```

A DO FOR ALL index must be of the form I,I+C, or I-C when used in a subscript. 2*I is not of one of these forms.

4)

```
        DO 1971 FOR ALL (I)/ P
   1971 A(I,I) = 0.0
```

Only a single subscript in an array reference may depend upon I.

5)

```
        DO 515 FOR ALL (J)/Q
        IF(A(J)) 2,2,1
      2 A(J) = 0.0
        GO TO 515
      1 A(J) = A(J)/B(J)
    515 CONTINUE
```

A DO FOR ALL index may not appear in any statement except for a DO FOR ALL assignment statement. The use of J in the arithmetic IF statement is therefore illegal.

6)

```
        DO 1 FOR ALL (I)/[1...50]
      1 A(J) = B(I)
```

The left-hand side of a DO FOR ALL assignment statement must use the DO FOR ALL index.

6.1.3 Input/Output Statements

There are three types of input/output statement:

1) READ and WRITE statements
2) ENCODE and DECODE statements
3) Auxiliary input/output statements

The first type consists of the statements that cause transfer of records between files and internal storage. The second type consists of statements which cause conversion of data within internal storage. They are classed with the input/output statements because the conversion they perform is identical with that performed for formatted READ and WRITE statements. The third type of statement consists of the BACKSPACE, REWIND, and FIND statements which provide for positioning a file and the ENDFILE statement which provides for demarcation of an external file.

The forms of the input/output statements are given in Table 7.

6.1.3.1 Transmission Options

There are three transmission options for use in Input/Output statements:

1) Record input/output
2) Formatted input/output
3) Namelist input/output

The transmission options specify the manner in which data is transformed during transmission.

6.1.3.1.1 Record Input/Output

For record input/output, data is transmitted without being transformed. Each input or output statement causes a single record to be transmitted. Each record consists of a string of data values in internal representation.

Execution of a record READ statement causes a record to be read from the specified input unit, and if an input list is specified, the values

TABLE 7: INPUT/OUTPUT STATEMENTS

READ/WRITE Statements

Type of Access	Type of Transmission	Form of READ	Form of WRITE
sequential	record	READ(u)	
		READ(u,END=g)	
		READ(u,ERR=h)	
		READ(u,END=g,ERR=h)	
		READ(u,ERR=h,END=g)	
		READ(u)K	WRITE(u)L
		READ(u,END=g)K	
		READ(u,ERR=h)K	
		READ(u,END=g,ERR=h)K	
		READ(u,ERR=h,END=g)K	
	formatted	READ(u,f)	WRITE(u,f)
		READ(u,f,END=g)	
		READ(u,f,ERR=h)	
		READ(u,f,END=g,ERR=h)	
		READ(u,f,ERR=h,END=g)	
		READ(u,f)K	WRITE(u,f)K
		READ(u,f,END=g)K	
		READ(u,f,ERR=h)K	
		READ(u,f,END=g,ERR=h)K	
		READ(u,f,ERR=h,END=g)K	
	namelist	READ(u,n)	WRITE(u,n)
		READ(u,n,END=g)	
		READ(u,n,ERR=h)	
		READ(u,n,END=g,ERR=h)	
		READ(u,n,ERR=h,END=g)	

Table 7 (Continued)

Type of Access	Type of Transmission	Form of READ	Form of WRITE
direct	record	READ(u'r)	
		READ(u'r,ERR=h)	
		READ(u'r)K	WRITE(u'r)L
		READ(u'r,ERR=h)K	
	formatted	READ(u'r,f)	WRITE(u'r,f)
		READ(u'r,f,ERR=h)	
		READ(u'r,f)K	WRITE(u'r,f)L
		READ(u'r,f,ERR=h)K	
	namelist	not permitted	

ENCODE/DECODE Statements

ENCODE(c,f,v)	DECODE(c,f,v)
ENCODE(c,f,v)K	DECODE(c,f,v)L

Auxiliary Input/Output

REWIND u
BACKSPACE u
FIND u'r
ENDFILE u

Notes:

1) u is an integer scalar expression which designates the input/output unit (file) to be used.

2) g is the statement label for an executable statement in the same program unit as the READ statement in which it appears. Transfer will be made to that statement if the end of the file is detected while reading.

Table 7 (Continued)

3) h is the statement label for an executable statement in the same program unit as the READ statement in which it appears. Transfer will be made to that statement if an error is detected while reading.

4) f is either the label of a FORMAT statement or the name of a singly-dimensioned array containing format information.

5) n is the name of a namelist established in a NAMELIST statement.

6) r is an integer scalar expression which specifies the record number for direct access input/output.

7) c is an integer scalar expression which gives the number of characters in v.

8) v is the name of a singly-dimensioned array containing a character string.

9) K is an input list.

10) L is an output list.

read are assigned to the sequence of elements specified by the list. The sequence of values required by the input list may not exceed the sequence of values from the record.

Execution of a record WRITE statement causes a record consisting of values obtained from the output list to be written upon the specified output unit. If the record did not previously exist, a new record is created.

<u>Examples</u>:

```
READ (3) A(I),B,C(I,*)
WRITE(UNITS) A(I,*)*B(*,I),1.3,SIN(E)
```

6.1.3.1.2 Formatted Input/Output

Under formatted Input/Output, data is transmitted under control of a FORMAT statement (6.2.3) which specifies the manner in which internal data is to be transformed from or to a character string respectively. Each input/output statement causes one or more records to be transmitted. Each record consists of a string of characters and appears on a separate line when printed.

Execution of a formatted READ statement causes input of one or more records from the specified unit. The information is scanned and converted as specified by the indicated format statement or format array. The resulting values are assigned to the elements specified by the list.

Execution of a formatted WRITE statement causes the values specified in the output list to be converted according to the format specification and written as one or more adjacent records on the specified unit.

<u>Examples</u>:

```
77  FORMAT(4E14.4)
    READ (4,77)A,B,C,D
    WRITE(7,77)A+B,B-C,C*D,D**A
```

6.1.3.1.2.1 ENCODE and DECODE Statements

The ENCODE and DECODE statements transmit data between an input/output list and the first c characters of a singly-dimensioned array variable under control of a format statement or format array.

The ENCODE statement converts data from the singly-dimensioned array variable into internal form and assigns the converted values to the input list items. The input list and format statement must not specify that more characters be converted than are specified in the character count c. If fewer characters are called for than are specified by c, the remainder are ignored. The character slash in the FORMAT statement has no effect for ENCODE.

The DECODE statement converts data in internal form from the output list to a character string which is placed in the singly-dimensioned array. The input list and format statement must not specify that more characters be converted than are specified by the character count c. If fewer characters are called for than are specified by c, blanks are placed in the remaining character positions. The character slash in the FORMAT statement has no effect for DECODE.

6.1.3.1.3 Namelist Input/Output

Under namelist input/output, data is transmitted under control of a NAMELIST statement (6.2.3) which specifies the names of data to be transferred. Both the name of the data and its value appear in the character string which is read from or written to the input/output unit. Each input/output statement causes one or more records to be transmitted. Each record consists of a string of characters and appears on a separate line when printed.

6.1.3.1.3.1 Namelist Input

Input data must be in a special form in order to be read using a NAMELIST list. The first character of each record to be read is ignored and will usually be blank. The second character in the first record of a group of records must be a $ or an & followed immediately by the NAMELIST name. The NAMELIST name must be followed by a blank and must not contain any embedded blanks. This name is followed by data to be read and converted. The end of the data group is signalled by a $ or an & either in the same record as the NAMELIST name, or as the second character of any succeeding record. The remainder of the record following the terminal $ or & is ignored. Data items must be separated by commas and be of the following form:

$$S = K$$

or

$$A = K_1, K_2, \ldots, K_m$$

where S is a scalar variable name or an array element reference, A is an array variable name or array cross-section, and each of the K's is a constant of integer, double integer, real, double precision, complex, double complex, or logical data type. Logical constants may be written in the form T, .T., or .TRUE. and F, .F., or .FALSE. A series of r identical constants may be represented by r*k where r is an integer constant and k is the repeated constant. Logical, complex, and double complex constants must be associated with variables of identical type. The other types of constant (integer, double integer, real, double precision) may be read into any type of variable (except logical, complex, and double complex) and will be converted to the type of the variable.

The variable names specified in the input data must appear in the NAMELIST list, but the order is not significant.

Embedded blanks are not permitted in constants. Trailing blanks after integers and exponents are treated as zeros.

Example:

```
        REAL A(3)
        LOGICAL EE
        COMPLEX III(5)

        NAMELIST / ROSE/ A,EE,III
        .
        .
        .
        READ (4, ROSE)
```

could read input data of the form:*

```
        Δ&ROSEΔA=3,5.6,4,EE= .TRUE.,III=4*(1.3,-4.2),(0.,0.)
        Δ &END
```

6.1.3.1.3.2 Namelist Output

When a namelist WRITE statement is executed, all variables and arrays specified in the associated NAMELIST statement are output. An array is output with its leftmost subscript varying most rapidly. The output data is written so that the data fields are large enough to contain all significant digits and so that the output can be read using namelist input.

6.1.3.2 Access Options

There are two access options for use in input/output statements:

1) Sequential access
2) Direct access

* The character Δ represents the character blank.

6.1.3.2.1 Sequential Access

Sequential Access files permit records to be written and read only in sequence from first to last. A sequential READ or WRITE statement processes the record or records which immediately follow the record last processed.

In addition to sequential READ and WRITE statements, the REWIND, BACKSPACE and ENDFILE statements may be applied to sequential files.

The REWIND statement causes a subsequent sequential READ or write statement to read from or write to the first record of the specified unit. Not all units can be rewound (e.g., printers, card readers). For these units the REWIND statement is ignored.

The BACKSPACE statement causes the specified unit to backspace one record. If the unit is already at its initial point, the statement has no effect. If the unit (e.g., a teletype) can not be backspaced, the statement has no effect.

The ENDFILE statement defines the end of a file of data on a unit by writing a unique record called an end-of-file record.

6.1.3.2.2 Direct Access

Direct access files permit records to be written and read in random order. A direct access input or output statement processes a specified record and those records which immediately follow it. In addition, it makes available the record number of the record which follows the last record processed, and so permits a form of sequential processing of records as a special case.

The DEFINE FILE statement (6.2.3) is required for each direct access unit. In it are defined the characteristics of the unit and an associated integer variable which is set to the number of the record following that last transmitted on the conclusion of each direct access READ or WRITE statement, and is set to the number of the record found at the conclusion of a FIND operation.

The FIND statement overlaps record retrieval from a direct-access unit with computation in the program, thereby increasing execution speed. The program has no access to the record that was found until a READ statement for that record is executed. (There is no advantage in having a FIND statement precede a WRITE statement.)

6.1.3.3 Input/Output Lists

READ, WRITE, ENCODE, and DECODE statements permit the use of an input/output list to specify the data values to be written or the data locations into which data is read.

6.1.3.3.1 Output Lists

An output list element is a

1) scalar expression
2) array expression
3) a parenthesized output list
4) an output implied DO of the form:

$$(L,\ i = m_1, m_2)$$

or

$$(L,\ i = m_1, m_2, m_3)$$

where L is an output list and i and the m's are the implied DO control variable and parameters, respectively.

An output list is one or more output list elements, separated by commas.

Examples:

```
A,C(I,*,J)
A,B(I,J),SIN(X)
(A(I),I=1,6)                (=A(1),A(2),A(3),A(4),A(5),A(6))
((B(I,J),J=1,2),I=1,2)      (=B(1,1),B(1,2),B(2,1),B(2,2))
3.5,5.7,-7.9,2**12
```

6.1.3.3.2 Input Lists

An input element is a

1) scalar variable name
2) array element reference
3) array variable name
4) array cross-section reference
5) a parenthesized input list
6) an input implied DO of the form:

$(L\ ,\ i = m_1,m_2)$

or $(L\ ,\ i = m_1,m_2,m_3)$

where L is an input list and i and the m's are the implied DO control variable and parameters, respectively.

An input list is one or more input list elements separated by commas.

Examples:

```
X(*,*,K), Y, Z(I)
(T(I,J),I=1,5,2)            (=T(1,J),T(3,J),T(5,J))
```

6.1.4 Debug Statements

The debug statements enable the user to locate errors in a IVTRAN source program. The debug statements provide for tracing flow within a program, tracing flow between programs, displaying the values of variables and arrays, and checking the validity of subscripts.

The debug statements consist of a DEBUG specification statement, an AT debug packet identification statement, the TRACE ON and TRACE OFF statements, and the DISPLAY debug output statement.

Debug statements are placed after the body of a program and before the END statement. This permits easy removal of the debug statements when debugging is complete. If debug statements are present they must appear in the following order:

1) DEBUG statement

2) One or more debug packets (if any) each consisting of an AT debug packet identification statement, followed by one or more executable debug statements (TRACE ON, TRACE OFF, DISPLAY) and other IVTRAN statements (executable, NAMELIST, FORMAT, and DATA statements.)

The program unit being debugged may not transfer control to any statement in a debug packet; however, the statements in the debug packet may transfer control to the program or return from it.

6.1.4.1 DEBUG Statement

The DEBUG statement is of the form:

DEBUG $S_1, S_2, \ldots, S_m$

where each S is one of the debug specifiers:

1) UNIT(u), where u is an integer constant specifying the output unit for debug output. If this option is not specified, debug output is placed on a standard output unit.

2) SUBCHK or SUBCHK $(m_1, m_2, \ldots, m_k)$, where the m's are array names. If this option is specified array subscripts are checked for validity. If the first form is used all array references are checked. If the second form is used, only references to the specified arrays are checked. If this option is not specified, no subscripts are checked.

3) TRACE. This specifier must appear if tracing is desired. If this option is omitted, no tracing will take place. Even when this option is used, a TRACE ON statement must be executed before tracing can commence.
4) INIT or INIT($m_1, m_2, \ldots m_k$), where each m is a variable or array name. When this option is specified, variables and array values are output when a statement which could modify its value (assignment, READ, and DECODE statements) is executed. If the first form is specified, all modifications are displayed. If the second form is specified only modifications to the named variables and arrays are displayed. If this option is not specified, no modifications are displayed.
5) SUBTRACE. If this option is specified, the subprogram name is displayed when the subprogram is entered and the message "RETURN" is displayed when control returns to the calling program.

Each of the debug specifiers may appear at most once. The debug specifiers may appear in any order.

Example:

```
        DEBUG     TRACE, SUBTRACE, UNIT(4)
```

6.1.4.2 AT Statement

An AT statement is of the form

```
        AT  k
```

where k is the statement number of an executable statement in the same program unit. The AT statement identifies the beginning of a debug packet and indicates the point in the program at which the debug packet is to be activated.

When control reaches the statement labelled k, control is transferred to the first executable statement following the AT statement. After the last statement of the debug packet is executed (provided it does not transfer control out of itself) control returns to the statement labelled k, which is then executed.

Example:

```
        .
        .
        .
  1     ABR = ACAD*AB+RA
  2     CALL DJINN (ABR)
        .
        .
        .
        DEBUG
        AT 2
        DISPLAY AB,ACAD,ABR,RA
        END
```

The assignment statement is executed first, followed by the DISPLAY statement and then the CALL statement.

6.1.4.3 TRACE ON Statement

The TRACE ON statement is of the form:

```
TRACE ON
```

The TRACE ON statement initiates the display of statement flow by statement number. Each time a labelled statement is executed, a record of the statement number is made on the debug output unit. This statement has no effect unless the TRACE specifier was used in the DEBUG statement. For a labelled statement which has a debug packet associated with it, the actions within the debug packet are executed before the label trace is output. Tracing continues through each level of subprogram call until a TRACE OFF statement is executed, provided the subprogram in question has the TRACE option specified in a DEBUG statement.

6.1.4.4 TRACE OFF Statement

The TRACE OFF statement is of the form:

```
TRACE OFF
```

Execution of the TRACE OFF statement suspends program flow tracing initiated by the TRACE ON statement.

6.1.4.5 DISPLAY Statement

The DISPLAY statement is of the form:

DISPLAY $m_1, m_2, \ldots, m_k$

whr ·e each of the m's is a non-dummy variable or array name. Tl.e DISPLAY statement outputs the values of the named variables and arrays on the debug output unit.

The effect of the DISPLAY statement is similar to the following two statements:

NAMELIST / m/ $m_1, m_2, \ldots, m_k$

WRITE (u,m)

where m is a namelist name not used elsewhere in the program and u is the debug output unit number.

Example:

DISPLAY PDA,PDQ,PDL

6.2 Specification Statements

There are five types of Specification statement:

1) Data attribute declaration statements
2) DATA statement
3) Input/output specification statements
4) Subprogram specification statements
5) FREQUENCY statement

6.2.1 Data Attribute Declaration Statements

There are seven types of data attribute declaration statement:

1) IMPLICIT statement
2) Type statement
3) DIMENSION statement
4) COMMON statement
5) OVERLAP statement
6) EQUIVALENCE statement
7) DEFINE statement

The extent of an array must be specified by using an extent specifier in a type statement, a DIMENSION statement, or a COMMON statement. The extent specifier is written:

$(d_1, d_2, \ldots, d_n)$

where the d's are the dimensions of an n-dimensional array.

The type of a variable may be specified through either the type statement, the IMPLICIT statement, or the use of the built-in type convention.

Relations between different variables are specified through the use of the EQUIVALENCE, OVERLAP, and DEFINE statements.

6.2.1.1 IMPLICIT Statement

The IMPLICIT statement is written in the form:

IMPLICIT $s_1, s_2, \ldots, s_n$

where each of the s's is an implicit specifier of the form:

$t(L_1, L_2, \ldots, L_m)$

where each t is one of the type declarators given in Table 8 and each of the L's is a single letter or a range of letters denoted by a pair of letters separated by a hyphen (minus sign). The first letter in a range must precede the second in the alphabet. The same letter may only be specified once within an implicit statement.

The IMPLICIT statement must be the first statement following the subprogram statement if present. There can be at most one IMPLICIT statement in a given program unit. The IMPLICIT statement declares the data type of variables within the program unit by specifying that variables beginning with the designated letters are of the designated type.

TABLE 8: TYPE DECLARATORS

Data Type	Type Declarators
integer	INTEGER, I, INTEGER*4
double integer	DOUBLE INTEGER, DI, INTEGER*8
real	REAL, R, REAL*4
double precision	DOUBLE PRECISION, DP, DOUBLE, REAL*8
complex	COMPLEX, C, COMPLEX*8
double complex	DOUBLE COMPLEX, DC, COMPLEX*16
logical	LOGICAL, L

Example:

The IMPLICIT statement:

```
IMPLICIT INTEGER(I-N),REAL(A-H,O-Z)
```

defines the same types for undeclared variables as the built-in typing convention.

The statement:

```
IMPLICIT DOUBLE COMPLEX(A-F),LOGICAL(X,Z)
```

specifies that variables beginning with the letters A,B,C,D,E, and F are to be of type DOUBLECOMPLEX and that variables beginning with the letters X and Z are to be of type LOGICAL unless they appear in a type statement. Variables which begin with other letters are typed according to the built-in typing convention.

The statement:

```
IMPLICIT DI(A-Z), R(K)
```

is invalid and should be written:

```
IMPLICIT DI(A-J,L-Z),R(K)
```

instead.

6.2.1.2 Type Statement

The type statement is of the form:

$t \quad s_1, s_2, \ldots, s_m$

where t is one of the type declarators given in Table 8 and each of the s's is a type specifier of one of the forms:

m

or

m e

where m is a variable, array or function name and e is an extent specifier.

The type statement declares each m to be of data type t and to be an array with the attached extent, if present. An identifier may appear in at most one type statement in a program unit. Typing in a type statement takes precedence over the built in type conventions and those established in an IMPLICIT statement.

Examples:

```
DOUBLE COMPLEX     CO,NU,ND,RUM
INTEGER  UNIT(40,30),ITEM(100)
REAL    FACT
DOUBLE  PRECISION  EXACT(133)
LOGICAL    L(10000)
DP   M,K,Z
```

6.2.1.3 DIMENSION Statement

A DIMENSION statement is of the form:

DIMENSION $m_1e_1, m_2e_2, \ldots, m_ke_k$

where each m is an array or function name and each e is an extent specifier. The DIMENSION statement specifies that each m is an array variable or array function with extent e.

Examples:

```
DIMENSION      HERCM(90,55,90)
DIMENSION      F(10),F1(400),F2(40,10)
```

6.2.1.4 COMMON Statement

The COMMON statement is written in the form:

$$\text{COMMON} \quad /b_1/\, s_{11}, s_{12}, \ldots, s_{1m} \ldots /b_k/\, s_{k1}, s_{k2}, \ldots s_{km}$$

where each of the b's is an identifier representing a COMMON block name or is blank. If b_1 is blank the slashes may also be omitted. Each of the s's is of one of the forms:

N

or

M e

where N is an array or scalar name, M is an array name, and e is an extent specifier.

The COMMON statement in IVTRAN is used to allow access to variables used by more than one program unit. In IVTRAN a COMMON block used by more than one program unit must be declared identically except for names in each of the program units. A variable in a block of COMMON may not be declared in two different program units unless the following match in both program units:

1) Its position in the COMMON block
2) Its data type
3) Its extent

The same number of variables must be declared for a given COMMON block in each program unit in which it is used. In addition, the OVERLAP statement (6.2.1.5) when referring to COMMON variables and arrays must cause identical overlapping to occur in each program unit.

If a COMMON block is declared more than once in a program unit, the effect is the same as a single COMMON statement which contained all the variables and arrays.

<u>Examples</u>:

Assume the following for all examples:

```
IMPLICIT I(I), R(R), DP(D), C(C), L(L)
```

<u>Valid</u> COMMON statements:

1) In program unit 1:

```
COMMON  /GLOBAL/  I1(1000),D,C(14,16)
```

In program unit 2:

```
DIMENSION  I2(1000)
COMMON  /GLOBAL/  I2,DI,CE(14,16)
```

2) In program unit 1:

```
COMMON  /STOCK/   I1,I2,I3 /BOND/ CC,LL(10)
```

In program unit 2:

```
COMMON  /BOND/ CQ  /STOCK/ II,IJ
COMMON  /BOND/ LE(10)
COMMON  /STOCK/ IJ
```

3) In program unit 1 (see 6.2.1.5 for OVERLAP statement):

```
COMMON  /PLACE/ DP(10), R(100), I(40)
OVERLAP((R), (DP,I))
```

In program unit 2:

```
COMMON  /PLACE/ DP1(10), R1(100), I1(40)
OVERLAP   ((I1,DP1),(R1))
```

<u>Invalid</u> COMMON statements:

1) In program unit 1:

```
COMMON  /LAW/ R(10), R1(10), I(10,10)
```

In program unit 2:

```
COMMON  /LAW/ R2(20),I1(100)
```

Items in COMMON must match exactly, not just in length.

2) In program unit 1:

```
COMMON  /NAIL/  R(40)
```

In program unit 2:

```
COMMON  /NAIL/  I(40)
```

Data types must match in corresponding items in COMMON.

3) In program unit 1 (see 6.2.1.5 for OVERLAP statement):

```
    COMMON /NOUN/ C(1000), R(500), I(500)
    OVERLAP ((C), (R,I)
```

In program unit 2:

```
    COMMON /NOUN/ C1(1000), R1(500), I1(500)
```

Overlap statements, if present in one program unit, must be present in all program units and must overlap identical sets of items.

6.2.1.5 OVERLAP Statement

The OVERLAP statement is of the form:

OVERLAP $(s_1, s_2, \ldots, s_n)$

where each s is an OVERLAP specifier of the form:

$(E_1, E_2, \ldots, E_m)$

and each E is an OVERLAP element of the form:

$(n_1, n_2, \ldots, n_k)$

and each n is an array or scalar variable name.

Either all of the variables in an OVERLAP specifier must be in the same COMMON block or none of them may be in COMMON. The order in which variables appear in OVERLAP elements is aribtrary. The order in which OVERLAP elements appear in OVERLAP specifiers is arbitrary. The same variable or array name may appear at most once in an OVERLAP statement.

Each OVERLAP specifier indicates sets of variables (overlap elements) which the compiler may cause to share storage. Each of the variables in an OVERLAP element can share storage with any of the variables in any other OVERLAP element in the same OVERLAP specifier. Variables declared in the same OVERLAP element do not share storage with one another.

Example:

Assume that in the initial part of a program, an array C with extent (100,100) is needed; in the final stages of the program C is no longer used, but arrays A and B with extents (50,50) and 100, resoectively, are used. Storage space can be saved by using the statement:

```
OVERLAP ((C),(A,B))
```

which permits the compiler to overlap part of the array C with part or all of arrays A and B.

6.2.1.6 EQUIVALENCE Statement

In IVTRAN the EQUIVALENCE statement is of the form:

EQUIVALENCE $E_1,E_2,\ldots,E_n$

where each E is an equivalence specifier of one of the forms:

$(s_1,s_2,\ldots,s_n)$

or

$(a_1,a_2,\ldots,a_n)$

or

$(e_1,s_1,s_2,\ldots,s_n)$

where each s is a scalar variable name, each a is an array variable name, and e is an array element of the form $a(c_1,c_2,\ldots,c_m)$ where each c is an integer constant and m is the dimensionality of a. All of the variables in an equivalence specifier share the same storage. Equivalence is mathematical equivalence if the items are of the same data type and storage equivalence otherwise.

See the OVERLAP statement (6.2.1.5) for broader storage equivalence capabilites and the DEFINE statement (6.2.1.7) for broader mathematical equivalence capabilities.

The same scalar variable may not appear more than once in an EQUIVALENCE statement. Only certain combinations of data types are permitted as indicated in the following table:

Data Type	Can be equivalenced to:
integer	integer, real
double integer	double integer, double precision, complex
real	integer, real
double precision	double integer, double precision, complex
complex	double integer, double precision, complex
double complex	double complex
logical	logical

At most one of the variables in an equivalence specifier may be declared in COMMON. If the third form is used, only the array element e may belong to a COMMON block.

Examples:

Assume for all examples the following declaration:

```
IMPLICIT I(I),DI(J),R(R),DP(S),C(C),DC(D),L(L)
```

Valid EQUIVALENCE statements:

```
1)      DIMENSION  I(2,2)
        EQUIVALENCE (I(1,1),IXX),(I(1,2),IXY),(I(2,1),IYX),(I(2,2),IYY)
2)      DIMENSION I(40),R(40),J(100),S(100)
        EQUIVALENCE (I,R),(J,S)
        COMMON  /SENSE/I,S
3)      EQUIVALENCE (IO,I0),(DII,D11)
4)      COMMON  /WEALTH/I(100),JK1
        EQUIVALENCE (I(1),I1,II),(JK,JK1,J)
```

Invalid EQUIVALENCE statements:

```
1)      EQUIVALANCE (I2,J2)
```

Incompatible data types.

```
2)      COMMON  /CAUSE/I1(40),I2(4,10)
        EQUIVALENCE  (I1,I2)
```

a) Two arrays in COMMON may not be equivalenced to one another.

b) The extents of two equivalenced arrays must be identical.

3)
```
COMMON /TIME/ L1
DIMENSION L2(100)
EQUIVALENCE (L2(19),L1)
```
A logical array element may not be equivalenced to a COMMON scalar.

4)
```
COMMON /ROOM/ RING(41)
EQUIVALENCE (RING(1),RING(41)
```
Two array elements may not be equivalenced to one another.

5)
```
EQUIVALENCE (I1,I2), (I2,I3)
```
The same scalar or array name may not appear twice in the same or different equivalence statements.

6.2.1.7 DEFINE Statement

The DEFINE statement is of the form:

DEFINE $a_1e_1 = b_1, a_2e_2 = b_2, \ldots, a_ke_k = b_k$

where each of the a's is the array name of the item being defined, each of the e's is the extent of the array, and each of the b's is a base item of the form:

$n(p_1, p_2, \ldots, p_m)$

where n is the name of the base array and each of the p's is a subscript expression of one of the forms:

c

or

$k

or

$k+c

or

$k-c

where c is an integer constant and k is an integer constant between one and the dimensionality of a.

The DEFINE statement declares one or more arrays with extent e which are based upon arrays having storage. Each occurrance of a reference a $(s_1, s_2, \ldots s_n)$ to a defined array can be replaced by an equivalent reference n $(t_1, t_2, \ldots, t_m)$ to the base array, where each t_i equals p_i with \$k replaced by s_k.

Each \$k from \$1 to \$n must appear exactly once within the base item. The defined array name, a, must not appear in any other specification statement. The defined array has the same data type as the base array.

<u>Examples</u>:

<u>Valid</u> DEFINE statements:

1) DEFINE ROW1(10) = A(\$1,1),ROW2 = A(\$1,2)

A reference ROW1(I) is equivalent to a reference A(I,1) and a reference ROW2(I) is equivalent to a reference A(I,2).

2) DEFINE OFFSET(40,40) = ARRAY(\$1+2,\$2-1)

A reference OFFSET (I1+3,I2+2) is equivalent to a reference ARRAY(I1+4,I2+1).

3) DEFINE TRANSP(100,100) = BASE(\$2,\$1)

A reference TRANSP(I,J) is equivalent to a reference BASE(J,I)

<u>Invalid</u> DEFINE statements:

1) DEFINE A(10,10) = B(\$2)

Both subscripts must be used in the definition; that is, both \$1 and \$2 must appear in the base item.

2) DEFINE DIAG(100) = ARRAY(\$1,\$1)

Each subscript may be used at most once in the base item.

3) COMMON /PEOPLE/ A

DEFINE A(100) = B(\$1+1)

The defined item, A, may not appear in any other specification statement.

6.2.2 <u>DATA Statement</u>

The DATA statement is of the form:

$$\text{DATA } v_1/d_1/,v_2/d_2/,\ldots,v_n/d_n$$

where each v is a list of scalar variable names, array element references, array cross-section references, or array names and each d is a list of optionally-signed constants (integer, double integer, real, double precision, complex, double complex, logical, Hollerith, octal, or haxedecimal) any of which may be preceded by r*, where r is an integer constant indicating the number of times the following constant is to be replicated.

The constants in the data list must match the items in the variable list in number and type, with the following exceptions:

1) An integer constant may be used to initialize a double integer variable or array element.

2) An octal or hexadecimal constant may be used to initialize an integer or double integer variable or array element.

3) A Hollerith constant may be used to initialize variables and array elements of any type but logical. The number of characters must match the data type as given in the following table:

data type	number of characters for scalar	number of characters for n element array
integer	1 to 4	4n-3 to 4n
double integer	1 to 8	8n-7 to 8n
real	1 to 4	4n-3 to 4n
double precision	1 to 8	8n-7 to 8n
complex	1 to 8	8n-7 to 8n
double complex	1 to 16	16n-15 to 16n
logical		none

4) A logical array constant may initialize a logical array or array cross-section.

Examples:

Assume for all examples the statement:

IMPLICIT I(I),DI(J),R(R),DP(S),C(C),DC(D),L(L)

Valid DATA statements:

1) DATA I1,I2,I3 / 44,35,26 / , J1,J2 / 2*0 /
I1,I2, and I3 are initialized to 44,35, and 26, respectively.
J1 and J2 are both initialized to zero.

2) DIMENSION R1(4), SA(40,40)
DATA R1(1),R1(3), SA / 4.0,5.0,160*1.000 /

3) DIMENSION L(128),L1[illegible])
DATA L / .T., .F., .T., 125*.F. / , L1 /[1,4...40]/

4) DIMENSION J(4)
DATA J / 26HMULTIPLE PROCESSOR SYSTEMS /

Invalid DATA statements:

1) DATA DC1, DC2 / (1.202,3.403) /
Number of constants must equal number of variables.
(1.2D2,3.404) is a single double complex constant.

2) DATA S5/ 2.5/
Data types must match. A real constant may not initialize a double precision variable.

3) DIMENSION L(128), LI(128)
DATA L,L1 / 8HABCDEFGH, 4*ZFFFFFFFF /
Neither Hollerith, Octal, nor Hexadecimal constants may be used to initialize logical data.

6.2.3 Input/Output Specification Statements

There are three input/output specification statements.

1) FORMAT statement
2) NAMELIST statement
3) DEFINE FILE statement

6.2.3.1 FORMAT Statement

The FORMAT statement is of the form:

FORMAT $(C_1, C_2, \ldots, C_n)$

where each C is one of the format codes:

r I w
r DI w
r F w.d
r DF w.d
r E w.d
r D w.d
r G w
r G w.d
r Z w
r O w
r L w
r A w
T p
s P
w X,

is a Hollerith constant, or is a repeated group of the form:

$$r\ (C_1, C_2, \ldots, C_k)$$

where:

r, an optional repeat count, is an integer constant indicating the number of times a format code or repeated group is to be used. If r is omitted the code or group is used once.

w is a non-zero integer constant specifying the width of a field in characters.

d is an integer constant that specifies the number of digits to the right of the decimal point.

p is a non-zero integer constant specifying a column position.

s is an optionally-signed integer constant specifying a scale factor.

The FORMAT statement is used in conjunction with the formatted READ and WRITE statements and the ENCODE and DECODE statements. The FORMAT statement specifies the type of conversion to be performed for each item in an input or output list.

6.2.3.1.1 General Rules for FORMAT Statement

1) FORMAT statements must be labelled. The label of a FORMAT statement may only be referred to in a READ, WRITE, ENCODE, or DECODE statement.

2) A comma separating two format codes may be replaced by a series of one or more slashes. Each slash indicates the end of the current record and the beginning of a new record. A series of one or more slashes may precede the first format code or may follow the last format code in a FORMAT statement. In either case, each slash ends the current record and begins a new record.

3) The comma is optional following the P and X format codes, the count-delimited Hollerith constant, and the repeated group.

4) A complex or double complex output list item requires a format code to convert the real part and a second format code to convert the imaginary part.

5) When formatted output is prepared for printing, the first character of each record is not printed but has the following interpretation:

Character	Interpretation
blank	Advance one line before printing
0	Advance two lines before printing
1	Advance to first line of next page
+	No advance

For output to other units, the first character of the record is treated as data.

6) There are two types of format codes: data codes which correspond to input/output list items and non-data codes which are processed between the processing of input/output list items. The data codes include I, DI, F, OF, E, D, G, Z, O, L, and A formats. Non-data codes written after the last used data code are processed up to either the next data code or the final right parenthesis, whichever occurs first.

7) If there are more data codes than input/output list items, the remainder of the FORMAT statement is ignored. If there are more list items than data codes, the FORMAT statement is rescanned beginning with the repeated group terminated by the right-most right parenthesis, or if exists, with the beginning of the FORMAT statement. When rescan occurs, the current record is ended and a new record begun.

Examples:

```
a)    20 FORMAT (I2,I3)
      is equivalent to
      20 FORMAT (I2,I3/ I2,I3/ I2,I3/ .../ I2,I3)
b)    30 FORMAT (2X,2(I4,3(4X,I4)))
      is equivalent to
      30 FORMAT (2X,2(I4,3(4X,I4)) / 2(I4,3(4X,I4))/ ... )
```

8) An array may be used instead of a FORMAT statement. The content of this array may be initialized by a DATA statement or a READ statement, for example. The contents of the array is a character string in the same form as a FORMAT statement, except that the word FORMAT and the statement number are omitted.

Example:

```
DIMENSION A(2)
DATA  A/ 8H(2X,I10) /
.
.
.
READ (4,A) K
```

6.2.3.1.2 Non-data Codes

There are five non-data codes:

1) Tp
2) sP
3) wX
4) Hollerith Constant
5) Slash

6.2.3.1.2.1 Tp Code

The T format code specifies the character position in the record where transfer of data is to begin or continue.

Example:

```
        READ(IUN,40) I,J,K
  40    FORMAT (T20,I5,T10,I2,T60,I5)
```

will cause I to be read from characters 20 to 24 of the record, J from 10 to 11 and K from characters 60 to 65 of the record.

6.2.3.1.2.2 sP Code

The P format code specifies a positive, negative, or zero scale factor for use on real and double precision data with E, D, F, DF, and G data format codes. The effect of the scale factor for input and output is

$$\text{external number} = \text{internal number} \times 10^{s}$$

A scale factor remains in effect until the end of the input/output statement or until superseded by another sP code.

Input: A scale factor may be specified for any real data, but takes effect only if an exponent is not specified in the input record.

Examples:

Code	Input	Internal Value
-2PF7.4	1.0E2	100.0
-2PF7.4	12.34	1234.
3PF7.4	1.0E2	100.0
3PF7.4	12.34	.01234

Output: A scale factor can be specified for real numbers output with or without exponents. For numbers without exponents the relation between internal value and external value is the same as for input. For numbers output with exponents, the decimal point is moved and the exponent adjusted to account for it.

Examples:

Code	Internal Value	Output
F9.4	12.34	12.3400
2PF9.4	12.34	1234.0000
-2PF9.4	12.34	.1234
E12.3	3928.6	0.393E+04
2PE12.3	3928.6	39.286E+02
-2PE12.3	3928.6	0.004E+06

6.2.3.1.2.3 wX Code

The X format code skips w characters on input and writes w blanks on output.

Example:

```
      WRITE (I,50) I,J,K
   50 FORMAT (I5,10X,I5,10X,I5)
```

cause I to be written in character positions 1 to 5, J in 16 to 20, K in 31 to 35 and blanks in positions 6 to 15 and 21 to 30.

6.2.3.1.2.4 Hollerith Constant

Both count-delimited and quote delimited Hollerith constants are permitted in FORMAT statements. The data is read or written directly to or from the FORMAT statement. If a quote-delimited Hollerith constant is used, an apostrophe in the data is represented as two apostrophes.

Input: Information read from the input record replaces the characters of the Hollerith constant.

Example:

```
400 FORMAT ('HOLLERITH')
    READ (7,400)
```

Nine characters are read from the input record and replace the characters H-O-L-L-E-R-I-T-H.

Output: The constant is written on the output record.

Example:

```
        1000  FORMAT(13H1PAGEΔHEADING)
              WRITE(4,1000)
```

The thirteen characters following the H are written on the output record. If the record is printed, the first character will cause skipping to the top of a new page.

6.2.3.1.2.5 Slash

The slash specifies the end of a record on input or output.

Input: The remainder of the current record is ignored and further input begins with the first character of the next record. Initial, final, and adjacent slashes cause skipping of whole records.

Output: The current record is terminated and a new record begun. Initial, final, and adjacent slashes cause blank records to be written.

Example:

```
              WRITE(3,17)I,J
         17   FORMAT (5HLINE1//5HLINE3)
      cause the following output:
              LINE1
              (blank line)
              LINE3
```

6.2.3.1.3 Data Codes

There are eleven data format codes:

1) Iw
2) DIw
3) F w.d
4) DFw.d
5) E w.d
6) D w.d
7) Gw and G w.d
8) Zw
9) Ow
10) Lw
11) Aw

Each data format code corresponds to an item in an input or output list and specifies the form of the corresponding data field in a record. If w characters are insufficient to hold a number on output, the field is filled with asterisks.

6.2.3.1.3.1 Iw and DI w Codes

I and DI format codes are used for transmitting integer and double integer data, respectively.

Input: The input field consists of w decimal digits and blanks. Embedded and trailing blanks are interpreted as zeros.

Output: The number is output right justified in a field of w characters, with leading blanks.

Example:

```
      WRITE(5,6)432
    6 FORMAT(I4)
```

cause Δ432 to be written.

6.2.3.1.3.2 Fw.d and DFw.d Codes

The F and DF codes are used for transmitting real and double precision data, respectively.

Input: Input is in one of the following forms:

i	i.	i.f	.f
i+e	i.+e	i.f+e	.f+e
i-e	i.-e	i.f-e	.f-e
iEe	i.Ee	i.fEe	.fEe
iE+e	i.E+e	i.fE+e	.fE+e
iE-e	i.E-e	i.fE-e	.fE-e
iDe	i.De	i.fDe	.fDe
iD+e	i.D+e	i.fD+e	.fD+e
iD-e	i.D-e	i.fD-e	.fD-e

where i, f, and e are strings of decimal digits representing the integer, fraction, and exponent parts of a real number. If the decimal point is not specified, the decimal point is assumed to be d digits from the right hand side of i. In other words, the internal value is 10^{-d} times the external value. A scale factor (6.3.1.2.2) applies only if e is not specified.

Examples:

Format Code	Input	Internal Value
F5.2	Δ1.23	1.23
F5.2	1.23Δ	1.23
F5.2	ΔΔ123	1.23
F5.2	Δ123Δ	12.30
1P F5.2	Δ1.20	0.123
1P F5.2	1.2+1	12.0

Output: The output is written as a sign (if negative), an integer part, a decimal point and d fractional digits right-justified in a field of w characters. If a scale factor (6.3.1.2.2) has been specified, it is applied.

Examples:

Format Code	Internal Value	Output
F5.2	0.001	Δ0.00
F5.2	0.01	Δ0.01
F5.2	0.12	Δ0.12
F5.2	1.23	Δ1.23
F5.2	12.34	12.34
F5.2	123.45	***** (overflow)
1P F5.2	.123	Δ1.23
-1P F5.2	.123	Δ0.01
F5.2	99.996	100.00
F5.2	-99.996	***** (overflow)
F5.2	-12.5	***** (overflow)
F6.2	-12.5	-12.50

6.2.3.1.3.3 Ew.d and Dw.d Codes

The E and D codes are used for transmitting real and double precision data, respectively.

Input: Input for Ew.d is identical to Fw.d input. Input for Dw.d is identical to DFw.d input.

Output: The output is written as a minus sign (if signed), an integer part, a decimal point, d fractional digits, and an exponent part, right-justified in a field of w characters. The form of the exponent part depends on the magnitude of the exponent as given in the following table.

Exponent Value	Exponent Form
0 to 9	E± 0e
10 to 99	E± ee
100 to 999	± eee
1000 to 9999	± eeee

If a scale factor (6.3.1.2.2) is specified, it changes both the exponent and the number of integer part digits.

Examples:

Format Code	Internal Value	Output	
E12.4	12.34	ΔΔ0.1234E+02	
2P E12.4	12.3456	Δ12.3456E+00	
-2P E12.4	12.34	ΔΔ0.0012E+04	
E12.4	12.34E+20	ΔΔ0.1234E+22	
E12.4	12.34E+300	ΔΔ0.1234+302	
E12.4	12.34E+400	Δ0.1234+4002	
E10.4	-0.01	**********	(overflow)
E11.4	-0.01	-0.1000E-01	

6.2.3.1.3.4 Gw and Gw.d Codes

The G format code provides for transmission of integer, double integer, real, double precision, and logical data according to the type specification of the corresponding variable in the input/output list.

Input: The action of G format for input is given by the following table:

I/O list data type	Equivalent format	
integer	Iw	(d ignored if present)
double integer	DIw	(d ignored if present)
real	Ew.d	
double precision	Dw.d	
logical	Lw	(d ignored if present)

Output: The action of G format for output is given by the following table:

I/O list data type	Equivalent format	
integer	Iw	(d ignored if present)
double integer	DIw	(d ignored if present)
real	Fw.s,4X or Ew.d	
double precision	DFw.s,4X or Dw.d	
logical	Lw	(d ignored if present)

For real and double precision data, the form of output depends upon the value of the number to be output. If the value is less than 0.1 or greater than or equal to 10^d, Ew.d or Dw.d format is used. Otherwise the number is output without an exponent, the action of the scale factor is suspended, and a total of d significant integer and fraction digits are output.

Examples:

Format Code	Value	Output
G11.4	0.0123	Δ0.1230E-01
G11.4	0.1234	Δ0.1234ΔΔΔΔ
G11.4	1.2340	ΔΔ1.234ΔΔΔΔ
G11.4	12.340	ΔΔ12.34ΔΔΔΔ
G11.4	123.40	ΔΔ123.4ΔΔΔΔ
G11.4	1234.0	ΔΔ1234.ΔΔΔΔ
G11.4	12340.0	Δ0.1234E+05
1P G11.4	12.340	ΔΔ12.34ΔΔΔΔ
1P G11.4	12340.0	Δ1.2340E+04
G5	12	ΔΔΔ12
G5	.TRUE.	ΔΔΔΔT
G10.4	-1.0	-1.000ΔΔΔΔ
G10.4	-0.1	********** (overflow)

6.2.3.1.3.5 Zw and Ow Codes

The Z and O format codes are used to transmit, from a field of w characters, hexadecimal and octal representations of data of integer, double integer, real, double precision, and logical data type.

Input: A within the input fields only the following characters are permitted:

O input: 0,1,2,3,4,5,6,7, and blank
Z input: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F, and blank

In either case embedded and trailing blanks are treated as zeros. If the value read is too large for the input data type, leading digits are lost. The number right-justified with leading zeros is the new internal value for the input list variable.

Output: The octal or hexadecimal number is output right-justified in a field of w characters.

Examples:

Format code	Value	Output
O2	43_{10}	53
Z2	43_{10}	2B
Z3	43_{10}	Δ2B

6.2.3.1.3.6 Lw Code

The L format code is used to transmit logical data.

Input: The first non-blank character in the field must be a T or an F. The remainder of the characters in the field are ignored. T represents true and F, false.

Output: A T or an F is placed in the output field, preceded by w - 1 blanks, for true and false output, respectively.

6.2.3.1.3.7 Aw Code

The A format code is used to transmit character data stored in variables of integer, double integer, real, and double precision data type. The number of characters which can be stored in a variable is given in the following table:

Data Type	Number of Characters
integer	4
double integer	8
real	4
double precision	8
complex	8 (must be output as 2 reals)
double complex	16 (must be output as 2 double precisions)
logical	none

Input: w characters are read and stored left-adjusted with trailing blanks. If w is greater than the number of characters which can be stored in the variable, the leftmost characters are lost.

Output: The characters in the variable are right-adjusted in a field of w characters with leading blanks. If there are more than w characters in the variable, the leftmost characters only are printed.

Example:

```
        DATA A/ 4HQRST /
        WRITE (5,15) A,A,A
   15   FORMAT (A3/A4/A5)
```

cause the following output:

```
        QRS
        QRST
        ΔQRST
```

6.2.3.2 NAMELIST Statement

The NAMELIST statement is of the form:

$$\text{NAMELIST} \;\; /n_1,/v_1 \quad /n_2/v_2 \;\ldots\; /n_m/v_m$$

where each n is a namelist name and each v is a list of scalar and array variable names. The NAMELIST statement is used with namelist input/output transmission (6.1.3.1.3).

Example:

NAMELIST /SOME/A,C,E /ALL/A,B,C,D,E

SOME and ALL are namelist names and each can be used in READ and WRITE statements.

6.2.3.3 DEFINE FILE Statement

A DEFINE FILE statement is of the form:

$$\text{DEFINE FILE} \;\; u_1(n_1,s_1,f_1,v_1),\ldots,u_m(n_m,s_m,f_m,v_m)$$

where:

u is an integer constant representing a direct-access input/output unit. Each direct access input/output unit must be declared in a DEFINE FILE statement.

n is an integer constant representing the number of records on unit u.

r is an integer constant representing the maximum record size and specifies a word count if formatted input/output is not used and specifies a character count if formatted or mixed input/output is used.

f is one of the characters E, U, or L indicating formatted, unformatted, or mixed transmission, respectively.

v is the name of an integer scalar variable. At the conclusion of each direct access input/output operation on unit u, v is set to the record number of the next record. At the conclusion of a FIND operation v is set to the number of the record found.

The DEFINE FILE statement is used with direct access input/output units. Its use is described in section 6.1.3.2.2.

6.2.4 Subprogram Specification Statement

There are four classes of Subprogram Specification Statements:

1) Subprogram header statements which describe the characteristics of the program unit in which they appear.
2) The EXTERNAL subprogram statement which describes the characteristics of a subprogram referenced in the program unit in which it appears.
3) The statement function definition which defines a function for use in the program unit in which the definition appears.
4) The ENTRY statement which defines an entry point to a function or subroutine.

6.2.4.1 Subprogram Header Statements

There are three subprogram statements:

1) SUBROUTINE statement
2) FUNCTION statement
3) BLOCK DATA statement

6.2.4.1.1 SUBROUTINE Statement

A SUBROUTINE statement is of one of the forms:

SUBROUTINE N

or

SUBROUTINE N $(s_1, s_2, \ldots, s_m)$

where N is the subroutine name and each s is an argument specifier of one of the forms:

```
v
v     VALUE
a
a     VALUE
p
*
```

where v is a scalar variable name, a is an array variable name, and p is a subprogram name. The various argument specifiers have the following interpretations:

1) Use of the word VALUE specifies argument passage by value, that is, storage is assigned for the variable or array in the subprogram. A value parameter may not be used to return a value to the calling program.

2) Writing a scalar or array variable name by itself specifies that it will be referred to by location. In reference by location, the subprogram reserves no storage for the dummy argument. The subprogram uses the corresponding actual argument each time the dummy argument is referenced.

3) Writing an asterisk specifies that the actual argument is an alternate return, which can be referenced with a RETURN i statement.

A SUBROUTINE statement is used to begin a subroutine subprogram.

6.2.4.1.2 FUNCTION Statement

The FUNCTION statement if of one of the forms:

FUNCTION f $(s_1, s_2, \ldots, s_m)$

t FUNCTION f $(s_1, s_2, \ldots, s_m)$

e FUNCTION f $(s_1, s_2, \ldots, s_m)$

te FUNCTION f $(s_1, s_2, \ldots, s_m)$.

where t is one of the type declarators given in Table 8, e is an extent, f is a function name, and each s is an argument specifier of one of the forms:

v

v VALUE

a

a VALUE

p

where v, a, and p are as described in Section 6.2.4.1.1.

A FUNCTION statement is used to begin a FUNCTION subprogram.

6.2.4.1.3 BLOCK DATA Statement

A BLOCK DATA statement is of the form:

BLOCK DATA

A BLOCK DATA statement is used to begin a block data subprogram.

6.2.4.2 EXTERNAL Statement

The EXTERNAL statement is of the form:

EXTERNAL $s_1, s_2, \ldots, s_m$

where each s is an EXTERNAL specifier of one of the forms:

n
n $(a_1, a_2, \ldots, a_k)$
n $(a_1, a_2, \ldots, a_k)$ s

where n is a function or subroutine name, each a is an argument specifier, and s is a side-effects specifier.

If the first form is used, actual and dummy arguments must match in data type for each reference to n. If the second or third form is used, an actual argument of any arithmetic data type (integer, double integer, real, double precision, complex, or double complex) can be matched with a dummy argument of any similar or dissimilar arithmetic data type and conversion will be performed automatically. Such converted arguments may only be passed by value (6.2.4.1.1) and may not be used to return values to the calling program.

If the first form is used actual and dummy arguments must match in extent. If the second or third forms are used to specify a scalar function one or more scalar dummy arguments may correspond to array actual arguments with identical extent. The result of the function has the same extent as the array actual arguments.

An argument specifier is of one of the forms:

LABEL
SUBROUTINE
t FUNCTION
te FUNCTION
t
t USED
t SET
t USED SET
t SET USED
t e
t e USED
t e SET
t e USED SET
t e SET USED

where t is a type declarator (Table 8), and e is an extent specifier.

The LABEL option specifies that the argument is an alternate return.

The SUBROUTINE option specifies that the argument is a subroutine name.

The FUNCTION option specifies that the argument is the name of a function with given type and extent.

The USED and SET options indicate that a given argument is input or output to the subprogram, respectively. If neither is specified, function arguments are assumed to be USED and subroutine arguments are assumed to be both SET and USED.

A side-effects specifier is one of the forms:

USES $(u_1, u_2, \ldots, u_k)$
SETS $(s_1, s_2, \ldots, s_k)$
USES $(u_1, u_2, \ldots, u_k)$ SETS $(s_1, s_2, \ldots, s_k)$

where each u is the name of a common block, scalar, or array whose value(s) is (are) used by subprogram n, and each s is the name of a common block, scalar, or array whose value(s) is (are) modified by subprogram n. In either case k may be zero, indicating that no variables are used or set. If the USES or SETS option is not specified, it is assumed that a function neither uses nor sets any common variables and that a subroutine can both use and set any common variable.

An EXTERNAL statement <u>must</u> be used in each of the following situations:

1) Any external subprogram name used as an actual argument must be declared in an EXTERNAL statement. Built-in functions need not be so declared.

<u>Example:</u>

```
EXTERNAL FUNC1, FUNC2
•
•
•
CALL      SUB (FUNC1)
CALL      SUB (FUNC2)
```

2) A subprogram which is to be referenced with actual arguments of different type than the corresponding dummy arguments must be declared in an EXTERNAL statement.

Example:

```
        EXTERNAL   FUNC (INTEGER, REAL)
        .
        .
        .
        Z = FUNC (Z,I)
```

is equivalent to:

```
        Z = FUNC (IFIX(Z),REAL(I))
```

3) A function which is expected to be referenced with array actual arguments corresponding to scalar dummy arguments must be declared in an external statement.

Example:

```
        DIMENSION  A(400), B(400), I(400), J(10,400)
        EXTERNAL FUNC (INTEGER,REAL)
        .
        .
        .
        A = FUNC (3,B)
        B = FUNC (I,A(N))
        A = FUNC (J(4,*),A+B)
```

In each case one or more arguments are arrays with extent (400). FUNC yields a result with extent (400).

4) If a function sets its arguments, or uses or sets common, it must be so declared in an EXTERNAL statement.

Example:

Called program:

```
        FUNCTION ICOUNT (A,B)
        COMMON /C/I,J
        I = I + 1
        IF (I.GE.J) A=B
        ICOUNT = I
        RETURN
        END
```

Calling program:

```
COMMON /C/I,J
EXTERNAL ICOUNT (REAL SET,REAL USED) USES (C) SETS(J)
.
.
.
B = (X-Y)**(J-I)
K = ICOUNT(X,Y)
C = (X-Y)**(J-I)
```

If the EXTERNAL statement were not present, the compiler could compute (X-Y)**(J-I) once, assuming that X, Y, J, and I would not be modified by ICOUNT. The EXTERNAL statement forces the expression to be computed twice.

It is desirable, though not required, to use an EXTERNAL statement in the following situations:

1) If a subroutine does not use or set common variables or certain arguments, declaration in an EXTERNAL statement can permit more extensive optimization.
Example:

```
EXTERNAL   SUB(REAL USED,REAL SET)USES( )SETS( )
.
.
.
A = SIN(X) + COS(Y)
CALL SUB(X,Y)
B = SIN(X) + COS(Y)
```

Given the EXTERNAL statement, the compiler may compute SIN (but not COS) once instead of twice giving the following equivalent code:

```
S = SIN(X)
A = S+COS(Y)
CALL SUB(X,Y)
B = S+COS(Y)
```

2) If errors in argument matching are anticipated, the presence of an EXTERNAL statement will allow the compiler to diagnose such errors and simplify debugging.

Example:

```
        EXTERNAL   SUB (REAL,LABEL)
        .
        .
        .
        CALL       SUB (3.5, 42)
        CALL       SUB (4.7, &100)
```

With the EXTERNAL statement the first CALL statement will be diagnosed. Without it the error in the first CALL statement will not be diagnosed by the compiler.

6.2.4.3 Statement Function Definition

A statement function definition is a statement of the form:

$f\ (d_1, d_2, \ldots, d_n) = e$

where f is the name of the function being defined, each d is a dummy argument, and e is an expression. Each dummy argument must be distinct from other dummy arguments in the same function definition but may be the same as dummy arguments in other definitions and the same as other variables in the program. The expression may contain references to the dummy arguments.

A reference to a statement function $f(a_1, a_2, \ldots, a_m)$ is equivalent to the expression e with all instances of a dummy argument d_i replaced with the corresponding actual argument a_i (converted to the type of d_i). A statement function may reference a previously defined statement function.

Examples:

```
        PHUNK (I,J) = A(I) - A(J)
        .
        .
        .
        B = PHUNK(K,L)
        C = PHUNK(M+3,9)*3.4
        D = PHUNK(R,C)
```

The last statements are equivalent to:

```
B = A(K) - A(L)
C = (A(M+3) - A(9))*3.4
D = A(IFIX(R)) - A(IFIX(C))
```

6.2.4.4 ENTRY Statement

The ENTRY statement is one of the forms:

ENTRY n

or

ENTRY $n(s_1, s_2, \ldots, s_m)$

where n is a subroutine name if the program unit began with a subroutine statement and is a function name of the same type and extent as f if the program unit began with a FUNCTION f statement. Each s is an argument specifier of one of the forms given in section 6.2.4.1.1 if in a subroutine and of one of the forms given in section 6.2.4.1.2 if in a function subprogram.

The ENTRY statement is not executable but is part of the procedure part of a program.

An entry point name may be referenced in the same manner as a function or subroutine name. When referenced the actual arguments or subroutine name. When referenced the actual arguments in the call are associated with the dummy arguments and control proceeds with the first executable statement following the ENTRY statement.

Arguments associated by VALUE at reference to one entry point retain their values for use at a latter entry point.

Example:

Called program:

```
        SUBROUTINE  SUB(A VALUE, B VALUE, C VALUE)
        DIMENSION   A(100),B(100),C(100),D(100)
        RETURN
        ENTRY  SUPER (D,I)
        D(I) = SIN(A(I)+B(I))*COS(A(I)-B(I))*C(I)/D(I)
        RETURN
```

Calling program:

```
        DIMENSION   R(100),S(100),T(100),U(100,100)
        .
        .
        .
        CALL  SUB (R, S, T)
        .
        .
        .
        DO   10   J = 1,100
        DO   10   K = 1,100
  10    CALL  SUPER (U(J,*),K)
        .
        .
        .
```

Use of an entry point in this case passes the arguments R, S, and T once instead of 10,000 times. On the other hand, the value parameters A, B, and C require 300 elements in SUB which would not have been required had they been passed by location on each call within the DO loop.

6.2.5 FREQUENCY Statement

A FREQUENCY statement is of the form:

FREQUENCY $s_1, s_2, \ldots, s_n$

where each s is a frequency specifier of the form:

L $(i_1, i_2, \ldots, i_m)$

where L is the statement label of a control statement and each i is the relative frequency of execution of a control transfer in the statement labelled L. The following statements may have the relative frequency of branches specified in a FREQUENCY statement. In each case the relative frequency of execution of the branch to label k_j is i_j.

1) GO TO $i,(k_1,k_2,\ldots k_n)$

2) GO TO $(k_1,k_2,\ldots,k_n),i$

3) IF (e) k_1,k_2,k_3

4) IF (e) S   In this case i_2 is the relative frequency of execution of statement S and i_1 is the relative frequency of non-execution of S. If S is itself a control statement i_1 is the relative frequency of non-execution of S and i_{j+1} is the relative frequency of execution of the label k_j.

5) IF (e) k_1,k_2

6) CALL S$(\ldots,\&k_2,\ldots,\&k_3,\ldots,\&k_n,\ldots)$. i_1 is the relative frequency of execution of the primary return from S. i_2 through i_n give the relative frequency of execution of the alternate returns.

7) DO k n=B,E,D. i_1 is the number of loop iterations = (E-B)/D+1.

The FREQUENCY statement is optional and when specified allows the compiler to produce more efficient code. If the frequency statement is not provided, the following assumptions are made:

1) DO loops with constant parameters are executed (E-B)/D+1 times, where B,E, and D are the beginning, final, and increment DO parameter values.

2) DO statements with variable parameters are assumed to specify 5 iterations.

3) Alternate returns in CALL statements are assumed to have zero frequency of execution.

4) All other conditional branches are considered equally likely.

<u>Example</u>:

```
         FREQUENCY 10(1,99999)
         .
         .
         .
         DO    1040  J = 1,100
         DO    1040  I = 1,1000
         .
         .
         .
   10    IF (F(I).NE.0)  GO TO 40
         P(I) = F(J)**2 + P(I)**2
   40    CONTINUE
         .
         .
         .
 1040    CONTINUE
```

If the frequency statement were not present, the compiler would assume that F(I).NE.0 was true half of the time and false half of the time. It would then remove P(J)**2 from the inner DO loop assuming that it would then be executed only 100 times and not 50,000 times. However, the FREQUENCY statement indicates that it will only be executed one time on the average and the compiler thus leaves it as is, letting P(J)**2 be executed once rather than 100 times.

7. PROGRAM UNITS AND PROGRAMS

7.1 Program Units

There are four types of program unit:

1) Main program
2) SUBROUTINE subprogram
3) FUNCTION subprogram
4) BLOCK DATA subprogram

7.1.1 Main Programs

A main program consists of statements written in the following order:

1) IMPLICIT (optional)
2) Specification statements (optional)
   a) Type statement
   b) DIMENSION
   c) COMMON
   d) OVERLAP
   e) EQUIVALENCE
   f) DEFINE
   g) DATA
   h) FORMAT
   i) NAMELIST
   j) DEFINE FILE
   k) FREQUENCY
3) Statement function declarations (optional)
4) Executable statements* (at least one of which must be present), FORMAT, NAMELIST and DATA statements (all optional).
5) Debug part (optional) consisting of a DEBUG statement optionally followed by one or more debug packets preceded by an AT statement and consisting of one or more of the following statements:

* Excepting TRACE ON, TRACE OFF, and DISPLAY.

5) a) Executable statements
   b) FORMAT, NAMELIST, and DATA
   c) TRACE ON, TRACE OFF, and DISPLAY
6) END.

7.1.2 Subroutine Subprogram

A subroutine subprogram consists of statements written in the following order:

1) SUBROUTINE
2) IMPLICIT (optional)
3) Specification statements (optional)
   a) Type statement
   b) DIMENSION
   c) COMMON
   d) OVERLAP
   e) EQUIVALENCE
   f) DEFINE
   g) DATA
   h) FORMAT
   i) NAMELIST
   j) DEFINE FILE
   k) FREQUENCY
4) Statement function declarations (optional)
5) Executable statements* (at least one of which must be present), FORMAT, NAMELIST, DATA and ENTRY statements.
6) Debug part (optional) consisting of a DEBUG statement optionally followed by one or more debug packets preceded by an AT statement and consisting of one or more of the following statements:
   a) Executable statements
   b) FORMAT, NAMELIST, and DATA
   c) TRACE ON, TRACE OFF, and DISPLAY
7) END

* Excepting TRACE ON, TRACE OFF and DISPLAY

A subroutine is referenced in a CALL statement (6.1.2.3). In each CALL the number of actual arguments must match the number of dummy parameters. The actual arguments permitted for each dummy argument are given in the following table.

TABLE 9: SUBROUTINE ARGUMENT MATCHING

Dummy Parameter	Actual Argument
VALUE[1] scalar variable unmodified scalar variable	scalar expression of same[2] data type Hollerith constant of same size[7]
modified scalar variable	scalar location[3] of same[2] data type
VALUE[1] array variable unmodified array variable	array expression of same[2] data type and extent
modified array variable	array location[4] of same[2] data type and extent[4]
subroutine name	subroutine name
function name	function name
alternate return dummy[5]	alternate return[6]

Notes

1) See section (6.2.4.1.1) for a discussion of VALUE parameters
2) See section (6.2.4.2) for a relaxation on data type matching
3) A scalar location is a scalar variable or array element
4) An array location is an array variable or array cross section
5) An alternate return is the character & followed by a label
6) See section (6.2.2) for size of Hollerith constants

7.1.3 Function Subprogram

A function subprogram consists of statements written in the following order:

1) FUNCTION
2) IMPLICIT (optional)
3) Specification statements (optional)
    a) Type statement
    b) DIMENSION
    c) COMMON
    d) OVERLAP
    e) EQUIVALENCE
    f) DEFINE
    g) DATA
    h) FORMAT
    i) NAMELIST
    j) DEFINE FILE
    k) FREQUENCY
4) Statement function declarations (optional)
5) Executable statements* (at least one of which must be present), FORMAT, NAMELIST, DATA and ENTRY statements.
6) Debug part (optional) consisting of a DEBUG statement optionally followed by one or more debug packets preceded by an AT statement and consisting of one or more of the following statements:
    a) Executable statements
    b) FORMAT, NAMELIST, and DATA
    c) TRACE ON, TRACE OFF, and DISPLAY
7) END

Within the subprogram, the function name which appears in the FUNCTION statement may be used as a variable. The last assignment made to that variable before return is the value of the function reference.

* Excepting TRACE ON, TRACE OFF and DISPLAY.

A function is referenced in a function reference of the form

$$f\ (a_1, a_2, \ldots, a_n)$$

where f is a function name and each a is an actual argument. In each function reference the number of actual arguments must match the number of dummy parameters. A function must have at least one argument. The actual arguments permitted for each dummy argument are given in the following table.

TABLE 10: FUNCTION ARGUMENT MATCHING

Dummy Parameter	Actual Argument
VALUE[1] scalar variable unmodified scalar variable	scalar[2] expression of same[3] data type Hollerith constant of same size[4]
modified scalar variable	scalar[2] location[5] of same[3] data type
VALUE[1] array variable unmodified array variable	array expression of same[3] data type and extent
modified array variable	array location[6] of same[3] data type and extent
subroutine name	subroutine name
function name	function name

Notes

1) See section (6.2.4.1.1) for a discussion of VALUE parameters
2) See section (6.2.4.2) for use of array arguments here
3) See section (6.2.4.2) for a relaxation of data type matching
4) See section (6.2.2) for size of Hollerith constants
5) A scalar location is a scalar variable or array element
6) An array location is an array variable or array cross section

7.1.3.1 Built-in Functions

In addition to functions which a user of IVTRAN may write, several predefined functions are available for use. These are given in Table 8. In each case no EXTERNAL statement is required to define argument types or side-effects.

TABLE 11: BUILT-IN FUNCTIONS

General Function	Function[3] Name	Definition [1,2]	Argument		Result Type
			No	Type	
Natural and common logarithm	ALOG DLOG	$y = \log_e x = \ln x$	1 1	Real Double Precision	Real Double precision
	CLOG CDLOG	$y = PV \log_e z$	1 1	Complex Double Complex	Complex Double Complex
	ALOG10 DLOG10	$y = \log_{10} x$	1 1	Real Double Precision	Real Double Precision
Exponential	EXP DEXP	$y = e^x$	1 1	Real Double Precision	Real Double Precision
	CEXP CDEXP	$y = e^z$	1 1	Complex Double Complex	Complex Double Complex
Square Root	SQ RT DSQRT	$y = \sqrt{x} = x^{1/2}$	1 1	Real Double Precision	Real Double Precision
	CSQRT CDSQRT	$y = PV \sqrt{z} = PV\ z^{1/2}$	1 1	Complex Double Complex	Complex Double Complex

TABLE 11: BUILT-IN FUNCTIONS

General Function	Function[3] Name	Definition [1,2]	Argument		Result Type
			No	Type	
Sine and cosine	SIN DSIN	y = sin x	1 1	Real Double Precision	Real Double Precision
	COS DCOS	y = cos x	1 1	Real Double Precision	Real Double Precision
	CSIN CDSIN	y = sin z	1 1	Complex Double Complex	Complex Double Complex
	CCOS CDCOS	y = cos z	1 1	Complex Double Complex	Complex Double Complex
Tangent and cotangent	TAN DTAN	y = tan x	1 1	Real Double Precision	Real Double Precision
	COTAN DCOTAN	y = cotan x	1 1	Real Double Precision	Real Double Precision
Arcsine and arccosine	ARSIN DARSIN	y = arcsin x	1 1	Real Double Precision	Real Double Precision
	ARCOS DARCOS	y = arcos x	1 1	Real Double Precision	Real Double Precision

TABLE 11: BUILT-IN FUNCTIONS

General Function	Function[3] Name	Definition [1,2]	Argument No	Argument Type	Result Type
Arctangent	ATAN DATAN	$y = \arctan x$	1 1	Real Double Precision	Real Double Precision
	ATAN2 DATAN2	$y = \arctan x$	2 2	Real Double Precision	Real Double Precision
Hyperbolic sine and cosine	SINH DSINH	$y = \frac{e^x - e^{-x}}{2}$	1 1	Real Double Precision	Real Double Precision
	COSH DCOSH	$y = \frac{e^x + e^{-x}}{2}$	1 1	Real Double Precision	Real Double Precision
Hyperbolic tangent	TANH DTANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1 1	Real Double Precision	Real Double Precision
Absolute value	IABS* IDABS* ABS* DABS*	$y = \mid x \mid$	1 1 1 1	Integer Double Integer Real Double Precision	Integer Double Integer Real Double Precision
	CABS CDABS	$y=\mid x\mid=(x_1^2+x_2^2)^{1/2}$	1 1	Complex Double Complex	Complex Double Complex

TABLE 11: BUILT-IN FUNCTIONS

General Function	Function[3] Name	Definition [1,2]	Argument		Result Type
			No	Type	
Error function	ERF DERF	$y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2}du$	1 1	Real Double Precision	Real Double Precision
	ERFC DERFC	$y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2}du=$ 1-erf x	1 1	Real Double Precision	Real Double Precision
Gamma and log-gamma	GAMMA DGAMMA	$y=\int_0^\infty u^{x-1}e^{-u}du$	1 1	Real Double Precision	Real Double Precision
	ALGAMA DLGAMA	$y=\log\int_0^\infty u^{x-1}e^{-u}du$	1 1	Real Double Precision	Real Double Precision
Maximum and minimum values	MAX0* AMAX0* MAX1* AMAX1* DMAX1*	$y=\max(x_1,x_2,\ldots,x_n)$	≥2 ≥2 ≥2 ≥2 ≥2	Integer Integer Real Real Double Precision	Integer Real Integer Real Double Precision
	MIN0* AMIN0* MIN1* AMIN1* DMIN1*	$y=\min(x_1,x_2,\ldots,x_n)$	≥2 ≥2 ≥2 ≥2 ≥2	Integer Integer Real Real Double Precision	Integer Real Integer Real Double Precision
Modular arithmetic	MOD* AMOD* DMOD*	y=remainder $\frac{x_1}{x_2}$	2 2 2	Integer Real Double Precision	Integer Real Double Precision

TABLE 11: BUILT-IN FUNCTIONS

General Function	Function[3] Name	Definition [1,2]	Argument No	Argument Type	Result Type
Truncation	AINT DINT INT IDINT	y=(sign x) ·n where n is the largest integer ≤x	1 1 1 1	Real Double Precision Real Double Precision	Real Double Precision Integer Integer
Float	FLOAT DFLOAT DDFLOT	Convert from integer to real.	1 1 1	Integer Integer Doubie Integer	Real Double Precision Double Precision
Fix	IFIX IDFIX	Convert from real to integer.	1 1	Real Double Precision	Integer Double Integer
Transfer of sign	ISIGN* SIGN* DSIGN*	$y=(\text{sign } x_2) \cdot x_1$	2 2 2	Integer Real Double Precision	Integer Real Double Precision
Positive Difference	IDIM* DIM* DDIM*	$y=x_1-\min(x_1,x_2)$	2 2 2	Integer Real Double Precision	Integer Real Double Precision
Change precision	SNGL* ISNGL*	Change from double to single precision	1 1	Double Precision Double Integer	Real Integer
	DBLE* IDBLE*	Change from single to double precision	1 1	Real Integer	Double Precision Double Integer

TABLE 11: BUILT-IN FUNCTIONS

General Function	Function[3] Name	Definition [1,2]	Argument		Result Type
			No	Type	
Obtain real and imaginary parts of complex	REAL* DREAL*	y=real part (z)	1 1	Complex Double Complex	Real Double Precision
	IMAG* DIMAG*	y=imaginary part (z)	1 1	Complex Double Complex	Real Double Precision
Build complex from real and imaginary parts	CMPLX* DCMPLX*	$y=x_1 + i\ x_2$	2 2	Real Double Precision	Complex Double Complex
Complex conjugate	CONJG* DCONJG*	$y=x_1 - i\ x_2$ where $z = x_1+i\,x_2$	1 1	Complex Double Complex	Complex Double Complex
Array summation	ISUM IDSUM RSUM DSUM	$y = \Sigma\ x_i$	1 1 1 1	Integer Array D.Integer Array Real Array D.Precision Array	Integer Double Integer Real Double Precision
Array product	IPROD IDPROD RPROD DPROD	$y = \Pi\ x_i$	1 1 1 1	Integer Array D.Integer Array Real Array D.Precision Array	Integer Double Integer Real Double Precision
Array maximum	IMAX IDMAX RMAX DMAX	y = maximum x_i	1 1 1 1	Integer Array D.Integer Array Real Array D.Precision Array	Integer Double Integer Real Double Precision

TABLE 11: BUILT-IN FUNCTIONS

General Function	Function[3] Name	Definition [1,2]	Argument		Result Type
			No	Type	
Array minimum	IMAX IDMAX RMAX DMAX	y = minimum x_1	1 1 1 1	Integer Array D.Integer Array Real Array D.Precision Array	Integer Double Integer Real Double Precision
Array logical and	ALL	y=true if all x_1=true y=false otherwise	1	Logical Array	Logical
Array logical or	ANY	y=true if any x_1=true y=false otherwise	1	Logical Array	Logical

Notes:

1) PV means principal value. That is, $-\pi \leq y \leq +\pi$

2) Z is a complex number of the form $x_1 + i\ x_2$

3) Functions compiled in-line are marked with an asterisk

7.1.4 Block Data Subprogram

A block data subprogram consists of statements written in the following order:

1) BLOCK DATA
2) IMPLICIT (optional)
3) Specification statements (at least one of which must be present)
    a) Type statement
    b) DIMENSION
    c) COMMON
    d) OVERLAP
    e) EQUIVALENCE
    f) DEFINE
    g) DATA
4) END

7.2 Programs

A IVTRAN program consists of a main program, with possible subroutine subprograms (optional), function subprograms(optional), a single block dat subprogram (optional), together with any built in functions required. A IVTRAN program is prepared by compiling its main program and subprograms and linking them with the linkage editor. The result of this process may then be loaded and executed on the ILLIAC-IV.

APPENDIX A: ALLOCATION

In order to utilize the parallel data paths available on the ILLIAC-IV, data must be properly arranged in storage. This arrangement in storage is called allocation. The body of this manual has described a fixed default allocation which permits parallel operations on any single index of an array. This appendix describes a rich set of optional allocations which permit parallel operations on a set of simultaneous indices, more efficient use of storage and so-called "scatter vectors".

Section A.1 describes the options available for allocating data. Section A.2 describes the various places where optional allocations may be specified. Section A.3 describes the operations which may be performed upon data with optional allocation.

A.1 Optional Allocations

The storage of the ILLIAC-IV is arranged so that each of the 64 processing elements (PEs) can directly access 1/64 of storage. PE_i can address locations a where a mod 64 = i. To allow operations in parallel on an array index the array must be so arranged that successive index values correspond to array elements accessible to successive PEs. Such an arrangement is called physical skewing. If,in addition,successive index values correspond to successive storage addresses, the index is known as a preferred index.

It is sometimes desirable that variations in an index value not select different PEs. Such an index is called an aligned index. An aligned index is useful with scatter vectors which select a different element out of each row (or column).

A set of indices written $(i_1, i_2, \ldots, i_k)$ and called a multi-index, may be treated as a single index. For example, in a three dimensional array, the third and first indices might be so treated, in which case successive values of the pair of indices (if physically skewed) correspond to successive PEs.

Example:

Multi Index	Extent	PE_0	PE_1	PE_2	PE_3	PE_4	PE_5
(1,2)	(2,3)	A(1,1)	A(2,1)	A(1,2)	A(2,2)	A(1,3)	A(2,3)
(2,1)	(2,3)	A(1,1)	A(1,2)	A(1,3)	A(2,1)	A(2,2)	A(2,3)

As is clear from the example, if an operation can be performed upon a multi-index $(i_1, i_2, \ldots, i_k)$ it can also be performed upon multi-indices (i_1), (i_1, i_2), ..., and $(i_1, i_2, \ldots, i_{k-1})$ because successive values of these multi-indices correspond to successive PEs. (i_2), (i_3), etc. cannot be used for parallel operations because successive values do not correspond to successive PEs.

A.1.1 Allocation Specifier

The allocation specifier is of the form:

$$[m_1, m_2 \ldots, m_j]$$

where each m is a multi-index specifier of one of the forms:

$(i_1, i_2, \ldots, i_k)$
$(i_1, i_2, \ldots, i_k)$
$ $(i_1, i_2, \ldots, i_k)$

where each i is an integer constant representing an index from 1 to n where n is the dimensionality of the array being allocated. The same index may appear but once in an allocation specifier. The first form of multi-index specifier specifies a physically-skewed multi-index. The second form specifies an aligned multi-index. The third form specifies a preferred multi-index. Only one preferred multi-index may be specified in an allocation specifier. The order in which indices are specified in a multi-index specifier is important. However, multi-index specifiers may appear in any order within an allocation specifier.

If one or more indices are unspecified in an allocation specifier, each is implied to be a physically-skewed index. Thus [(1,7), (6,4)] is equivalent to [(1,7), (6,4), (2), (3), (5)]. There must be at least one physically skewed or preferred multi index, specified or implied by each allocation specifier.

Examples:

Valid Allocation Specifiers

[(1,2)]
[(4)]
[(3), (7), (2)]
[$(1), (2)]
[#(3,4),.$(1,2)]

Invalid Allocation Specifiers

[(4000)]	4000> number of dimensions
[(1,2), (3,2)]	index 2 appears twice
[$(1), $(2)]	preferred multi-index appears twice
[#(2,1)]	no physicall, skewed index

A.1.2 Multi-index

A multi-index is a sequence of indices which is treated as a single index for purposes of allocation. An array A with one or more multi-indices is equivalent to another array A' with fewer dimensions.

Examples:

A extent	A allocation	A reference	A' extent	A' allocation	A' reference
(10,5)	[(1,2)]	A(I,J)	(50)	[(1)]	A'(I+(J-1)*10)
(10,5)	[(2,1)]	A(I,J)	(50)	[(1)]	A'(J+(I-1)*5)
(3,4,5)	[(1,3)]	A(I,J,K)	(4,15)	[(2)]	A'(J,I+(K-1)*3)
(3,4,5)	[(1,3,2)]	A(I,J,K)	(60)	[(1)]	A'(I+3*(K-1)+15*(J-1))

A <u>permissable multi-index</u> is either a specified or implied multi-index $(i_1, i_2, \dots, i_k)$ or a prefix of a permissable multi-index $(i_1, i_2, \dots, i_\ell)$, $1 \le \ell < k$.

A.1.3 Physical Skewing

A physically skewed index provides for parallel operations upon part of an array at the expense of storage efficiency. For example, the array A with extent (3,4) and allocation [(1),(2)] may be arranged in storage in one of the following two ways:

PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	...	PE_{63}
$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$				
	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$			
		$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$		

PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	...	PE_{63}
$A_{1,1}$	$A_{2,1}$	$A_{3,1}$					
	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$				
		$A_{1,3}$	$A_{2,3}$	$A_{3,3}$			
			$A_{1,4}$	$A_{2,4}$	$A_{3,4}$		

The first arrangement corresponds to [(1),$(2)] and the second to [$(1),(2)]. Unless other variables can be placed in the rows used by the array A, the space will be wasted. The compiler attempts to minimize the wasted space by fitting arrays together (as one trys to fit suitcases into a car trunk). If the arrays are of the wrong size or are physically skewed when no parallel operations are to be performed upon them a considerable amount of space may be wasted (as car trunk space is wasted if the suitcases are the wrong size and shape). Generally space utilization is improved if

1) Different arrays are of similar extent and allocation.

2) Dimensions are multiples of 64 (or somewhat less; 61 is nearly as good as 64).

3) Multi-indices are used where possible to combine small dimensions into a larger index. E.g., (3,3,6,20) is much more economically allocated [(1,2,3)] or [$(3.4)] than [(1),(2),(3),(4)].

4) Large arrays are given special attention. Small arrays will generally fill in the cracks around larger arrays.

A.1.4 Aligned Indices

An aligned index may be used for a non-parallel index to decrease the amount of computation required for array accesses or it may be used to implement "scatter-vectors".

A scatter vector is an integer array which is used to select a different element from each row (column) of an array. Given an array A(5,5)[#(1),(2)], a scatter-vector SV(5) = 2,4,3,3,1, and an array B(5), the statements

```
     DO 1 FOR ALL I/[1...5]
1    B(I) = A(SV(I),I)
```

are equivalent to the statements:

```
B(1) = A(2,1)
B(2) = A(4,2)
B(3) = A(3,3)
B(4) = A(3,4)
B(5) = A(1,5)
```

This is possible because the five selected elements of A are in adjacent PEs, as shown in the following diagram, where the selected elements are circled.

PE_0	PE_1	PE_2	PE_3	PE_4	...	PE_{63}
$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$\boxed{A_{1,5}}$		
$\boxed{A_{2,1}}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$		
$A_{3,1}$	$A_{3,2}$	$\boxed{A_{3,3}}$	$\boxed{A_{3,4}}$	$A_{3,5}$		
$A_{4,1}$	$\boxed{A_{4,2}}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$		
$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$		

It is illegal to align all indices of an array. This would give an allocation (were it allowed) for an array A(4)[#(1)] which is wasteful of space and serves no useful purpose:

PE_0	PE_1	...	PE_{63}
A_1			
A_2			
A_3			
A_4			

A.1.5 Physical Skewing for Different Data Types

The discussion thus far has assumed that elements of all data types occupy a full word. In actuality, logical elements occupy 1 bit, integer and real elements occupy 32 bits (1 half word), and double complex elements occupy 128 bits (2 full words). These data types require special treatment, as outlined in the following sections.

A.1.5.1 Logical Data

One physically skewed index (the preferred multi-index) is assigned successive bit positions in successive words. Index values 1-64 are in bits 0-63 of PE_0, index values 65-128 are in bits 0-63 of PE_1, etc. Other indices are skewed in the usual fashion. For some operations (e.g., combining two such indices) 64x64 = 4096 elements may be operated on in parallel. For other operations only 64 elements can be operated on in parallel.

A.1.5.2 Integer and Real Data

The first 64 elements of the preferred index occupy the inner half word of a row of 64 words. The next 64 elements occupy the outer half word of the same row. For some operations, 2x64 = 128 elements may be operated upon in parallel. For others only 64 elements can be operated on in parallel.

A.1.5.3 Double Complex Data

The real parts of the first 64 elements of the preferred index occupy a row of 64 words. The imaginary parts of the same elements occupy the next row of 64 words.

A.2 Use of the Allocation Specifier

The allocation specifier is used in the following contexts:

1) logical array constants
2) DIMENSION, COMMON, and type statements
3) FUNCTION and EXTERNAL statements

A.2.1 Logical Array Constant

An enumerated logical array constant with its extent, if any, may be followed by an optional allocation specifier. An iterated logical constant is one dimensional, may not have an allocation specifier and is assumed allocated [$(1)].

Examples:

Valid constants:

```
[(1,1),(2,2),(3,3)] [(2,1)]
[ ](5,5,5)[(3,1),$(2)]
```

Invalid constants:

```
[1,7...400] [(1)]
```

an interated constant may not have an allocation specifier

```
[(4,3,2),(3,2,1)] [(4,2)]
```

index (4) larger than number of dimension

A.2.2 DIMENSION, COMMON, and Type Statements

The DIMENSION, COMMON, and Type Statements each permit an allocation specifier to follow the extent if present.

Examples:

```
DIMENSION   A(40,346)[#(1),(2)]
INTEGER   X(2,2,15)[(1,2,3)],ICE(40,30,20)
COMMON  /COLD/ICE[(1),$(3,2)]
```

The allocation declarations for the variables of a common block must be identical for each program unit in which a common olock is used.

A.2.3 FUNCTION and EXTERNAL Statements

The allocation of the result of an array function may be specified by writing an allocation specifier after the extent in a FUNCTION statement. This allocation must be specified in a DIMENSION statement in each program unit in which the function is used. In addition, the function must be declared in an EXTERNAL statement to distinguish it from an array variable.

Example:

```
(10,10) [(1,2)] FUNCTION  FOO(I)
.
.
.
END
```

Calling program:

```
DIMENSION  FOO(10,10)[(1,2)]
EXTERNAL  FOO
.
.
.
ARRAY = FOO(3)
.
.
.
END
```

The external statement may specify the allocation of an argument by writing the allocation specifier after the extent in the external statement. The allocation of actual arguments must match the allocation of formal (dummy) parameters. The EXTERNAL statement is required if the allocation of an argument is other than the default allocation.

Example:

```
SUBROUTINE  TWO(A)
DIMENSION   A(40,40)[(1),#(2)]
.
.
.
END
```

Calling Program:

```
EXTERNAL  TWO (REAL(40,40)[(1),#(2)])
```

A.3 Use of Allocated Data

Optional allocations extend the DO FOR ALL statement, the set selector, array expressions, and array assignment.

A.3.1 DO FOR ALL Statement

A DO FOR ALL statement may be of the form:

DO k FOR ALL $(I_1, I_2, \ldots, I_k)/S_k$

where each I is an integer scalar variable called a control index and $(I_1, I_2, \ldots, I_k)$ is called the control multi-index. S_k is a k-dimensional logical array expression with allocation $[(1, 2, \ldots, k)]$.

Within the range of the DO FOR ALL the control indices may only be used in statements of the form:

p = e

or IF(f) p = e

where

1) p is an array element reference with a permissable multi-index (A.1.2) $(i_1, i_2, \ldots, i_k)$ and each subscript $S_{i_j} = I_j$ or $= I_j \pm C$ where C is an expression independent of the control indices. All other subscripts must be independent of the control indices.

2) e and f are each an expression which may or may not depend upon the control indices. Within e and f any array element references are either of the form 1) above or are independent of the control indices.

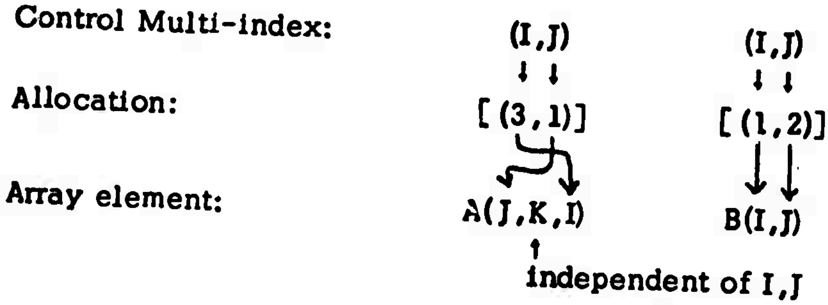
3) Within an array reference of the form 1) a subscript position corresponding to an aligned index may be of the form 2).

The execution of the range is identical to the description given in Section 6.1.2.7.2 except that each DO FOR ALL assignment is executed in parallel for all values of the multi-index as opposed to a single index.

Example:

```
      DIMENSION  A(8,40,8)[(3,1)],B(8,8)[(1,2)]
      LOGICAL    S(8,8)[(1,2)]
      .
      .
      .
      DO  1  FOR ALL (I,J)/S
 1    A(J,K,I) = B(I,J)
```

In this case the 64 assignments are carried out in parallel. Note the way the array elements A(J,K,I) and B(I,J) are constructed:



A.3.2 Set Selector

A set selector is written:

$$[(I_1,I_2,\ldots,I_k)/S_k:B]$$

where S_k and the Is are as described in Section A.3.1 and B is a boolean expression of the same form as the expression e in Section A.3.1. The result array expression has an extent $(d_{i_1},d_{i_2},\ldots,d_{i_k})$ where $(d_1,d_2,\ldots,d_m)$ is the extent and $(i_1,i_2,\ldots,i_k)$ is the permissable multi-index of S_k. The allocation of the result array expression is [(1,2,...,k)].

A.3.2 Set Selector

A set selector is written:

$$[(I_1,I_2,\ldots,I_k)/S_k:B]$$

where S_k and the I s are as described in Section A.3.1 and B is the boolean expression of the same form as the expression e in Section A.3.1. The result array expression has an extent $(d_{i_1},d_{i_2},\ldots,d_{i_k})$ where $(d_1,d_2,\ldots,d_m)$ is the extent and $(i_1,i_2,\ldots,i_k)$ is the permissable multi-index of s_k. The allocation of the result array expression is $[(1,2,\ldots,k)]$.

Example:

The array S in the example of Section A.3.1 could be initialized with a statement of the form:

S = [(I,J)/.NOT.[ ](8,8)[(1,2)] :I.NE.J]

A.3.3 Array Expressions

An array expression has, in addition to its extent, an allocation. The allocation of an expression is a set of permissable multi-indices (.1.2). The allocation of an array expression is determined as follows:

1) An array variable, constant, or function has the permissable multi-indices specified and implied by its allocation-specifier.

2) A parenthesized array expression, a unary array operator, or a binary operatcr operating on an array and a scalar has the same allocation as the array operand.

3) A binary operator combining two array expressions has an allocation which is the set of permissible multi-indices common to both operands. There must be at least one common multi-index for the expression to be legal.

4) A scalar function with one or more array actual arguments replacing scalar dummys has as an allocation the set of multi-indices common to all such arguments. There must be at least one common multi-index for the function call to be legal.

4) An array cross-section has the same allocation as the array with all permissable multi-indices omitted which have an index position which is not occupied by an asterisk. If there remain no multi-indices the cross-section is illegal. E.g., an array D with an allocation [(1),(2,3),(2)] has the following legal cross-sections:

cross-section	allocation
D(*,*,I)	[(1),(2)]
D(*,I,*)	[(1)]
D(I,*,*)	[(2,3),(2)]
D(*,I,J)	[(1)]
D(I,*,J)	[(2)]

D(I,J,*) is illegal because all multi-indices would be omitted.

Examples:

Assume the declarations

```
     DIMENSION  A(10,10)[(1),(2)],B(10,10)[(1,2)]
                C(10,10)[(2,1)],D(10,10,10)[(1),(2,3)]
     EXTERNAL   F(REAL,REAL)
```

Expression	Extent	Allocation
A	(10,10)	[(1),(2)]
B	(10,10)	[(1,2),(1)]
C	(10,10)	[(2,1),(2)]
A+B	(10,10)	[(1)]
A+C	(10,10)	[(2)]
B+C	illegal - no common multi-index	
D	(10,10,10)	[(1),(2,3),(2)]
D(*,*,I)	(10,10)	[(1),(2)]
D(*,*,I)+A	(10,10)	[(1),(2)]
F(D(*,*,I),C)	(10,10)	[(2)]

A.3.4 Array Assignment

Any array expression may be assigned to any array location with the same extent. If they share a permissible multi-index, the operation can be performed in a single parallel step. If they share no permissible multi-index, the compiler will use a temporary array which has one or more permissible multi-indices in common with the array expression and the array location.

Example:

```
DIMENSION     A(10,10)[(1),(2)],B(10,10)[(1,2)]
              C(10,10)[(2,1)]

A = B         can be performed in one parallel step using
              [(1)]
A = C         can be performed in one parallel step using
              [(2)]
B = C         must be performed (by the compiler) in two
              parallel steps:
              1)  t = C
              2)  B = t
                  where t has an allocation [(1),(2)]
```

CHAPTER III

FUNCTIONAL SPECIFICATION FOR THE ILLIAC IV LINK EDITOR

157-A

Introduction

The basic function of the linkage editor will be the linking of separately assembled ASK or FORTRAN compiled programs. Each assembly or compilation will generate an object module. The linkage editor, which operates from its own source language , is used to specify the relationships among these object modules. The output is a file, called a load module, which instructs the loader of the memory layout desired.

Although the linking of object modules is the primary function of the linkage editor, it will also accomplish all of the following items:

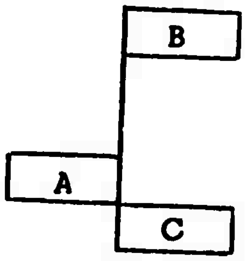
1) Text relocation,
2) Generation of a load module,
3) Library loading and linking,
4) Subsegment generation,
5) Issuing maps and error diagnostics,
6) Resolving external references and entry points,
7) Storage allocation,
8) Checking for consistency of data types between actual and formal parameters,
9) Checking for consistency among array declarations.

The proposed linkage editor permits the user to physically arrange the object modules in memory without imposing any logical relationships. In addition, it allows for the loading of modules without constraining the memory to hold unrequested modules. This concept is known as the "memory occupation specification" technique. The benefit of such a technique is that it eliminates the tree structure imposed by most linkage editors. Tree structuring implies that there is some logical relationship among object modules throughout the program. It also imposes a hierarchy, when often none exists, upon the functional operation of the modules.

Segmentation

To minimize the memory requirements, a programmer may organize his program into segments. Segments are constructed from separately generated object modules. A segment may have within itself other non-related segments which overlay one another (i.e., subsegments).

We shall illustrate a segmented program as shown below. In the diagram, a horizontal coordinate is used to denote increasing memory storage from left to right; a vertical coordinate is used to denote subsegments and disjoint programs. Thus, if a program is to consist of a portion A, which is to remain in memory at all times, and two portions B and C which follow A and are to be overlayed, we would use the diagram:



where B and C utilize the same memory. Segments and subsegments are defined by utilizing the SEG control statement.

The following is a syntactic specification for the SEG control statement.

```
seg-statement        ::=    SEG specification
specification        ::=    routine-name |
                            routine-name , segments |
                            common , routine-name |
                            common , routine-name, segments
segments             ::=    subsegment |
                            segments, subsegment
subsegment           ::=    routine-name |
                            ( overlay ) |
                            common , routine-name
overlay              ::=    overlay = segments |
                            segments
common               ::=    / ident / |
                            common , / ident /
routine-name         ::=    ident
ident                ::=    identifier
```

An "identifier" must begin with a letter and consist of no more than six letters or digits.

The following describes the terminals used in the specification portion of a SEG control statement:

, indicates that two programs are to occupy memory at the same time, and are to be contiguous,

= indicates that two programs are to begin at the same memory location (i.e., disjoint),

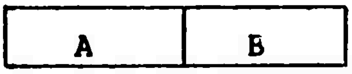
() indicates subsegments (i.e., grouping),

// indicates COMMON allocation.

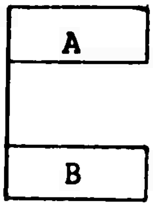
The terminal "," is considered to have greater precedence than the terminal "=".

The following is a set of examples to clarify the syntax.

The specification A, B would cause the following allocation of storage:

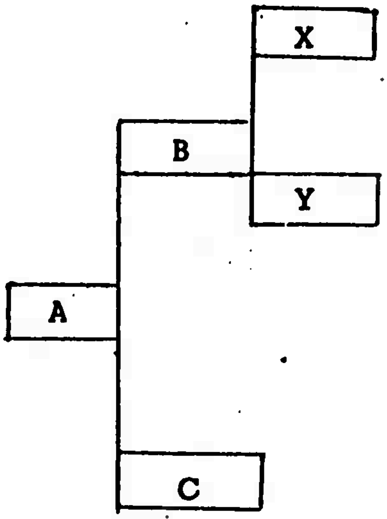


and the specification (A = B) would cause the following allocation of storage:



where A and B are disjoint.

The specification A, (B, (X = Y) = C) would cause the following allocation of storage:

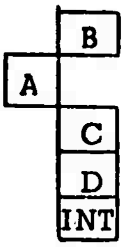


where (B = C) and (X = Y) are disjoint subsegments.

That is, either C or B may be in memory at one time and if B is, then either X or Y may also be in memory.

The following example illustrates the total elimination of any tree structure:

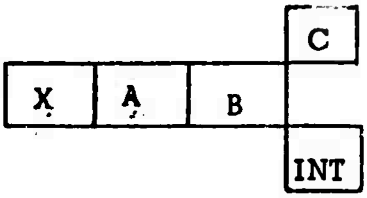
A, (B=C=D=INT)



where the segments B, C, D, and INT are overlayed. The appropriate segment is loaded when the routine is called.

The following example shows how FORTRAN library routines and COMMON could be allocated:

/X/A ,B, (C = INT)

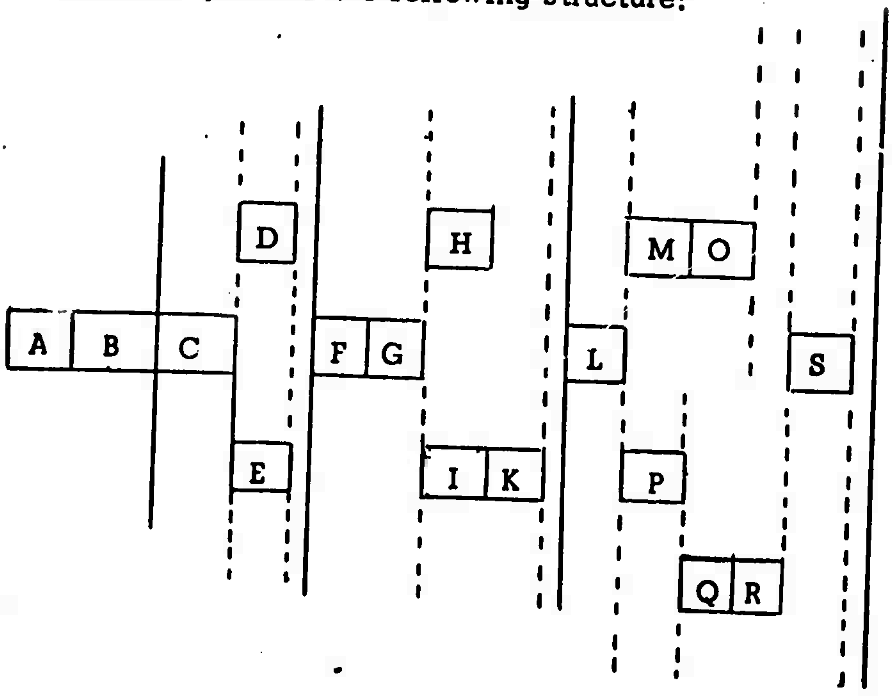


where the common area /X/ is associated with the segment A, B and the subroutines INT and C overlay one another. It is the user's responsbility to insure that the segment containing the common area /X/ is in memory when needed. If the location of the common areas or library routines are not specified, the linkage editor will allocate them to the first segment. Note, that an allocated common area, or a sequence of common areas, must be followed by a "," "routine-name". Tie common area(s) are not loaded until the associated routine is called.

A linking sequence could be:

```
SEG     A,B
SEG     C, (D=E)
SEG     F, G, (H=I, K)
SEG     L, (M, O=P, (Q,R)), S
```

which will produce the following structure:



where | denotes segments and ¦ denotes subsegments.

Note, that either D or E may be in memory at any one time and that a subsegment is constructed. Each segment or subsegment may call any other segment or subsegment that is not disjoint from the calling segment. The linkage editor will check that the modules of a subsegment are in fact disjoint.

In the above example, no relationship is implied between the modules. Modules may be allocated in any configuration that is reasonable to the user.

Using the Linkage Editor

The linkage editor is loaded by typing:

```
.RUN LINK
#file-name
```

where # indicates that the linkage editor is waiting for a file name. The indicated file will contain the control statements.

Control Statements

The source language input is discussed in this section. The control statements may start in any column. Following the mnemonic operation, and separated by at least one blank, is the specification portion. Names of modules and files in the specification portion must begin with a letter and consist of no more than six letters or digits. The terminal symbols may or may not be surrounded by blanks. Each control statement is discussed in this section.

The ENTRY statement (optional):

The ENTRY statement has the form:

```
ENTRY   symbol
```

where "symbol" is the name of an object module (i.e., subroutine) where execution is to begin. If the ENTRY statement is omitted, it is assumed that the first instruction in the first segment is the entry point.

The OMIT statement (optional):

The OMIT statement has the form:

$$\text{OMIT} \quad \text{symbol} \quad \{,\ \text{symbol}\}_{0}^{\infty}$$

where "symbol" specifies those external references that are not to be resolved by the linkage editor.

The SEG statement:

The SEG statement has the form:

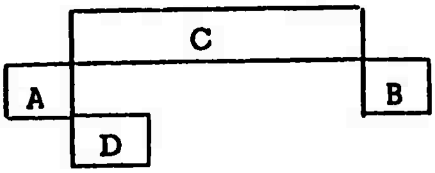
SEG specification

where the "specification" portion is the definition of the segment and subsegments being defined.

All subsegments must be enclosed in at least one level of parentheses. Consider the example:

SEG A, (C=D), B

which represents



where C and D overlay each other.

Initially, segment A is loaded into memory. Subsegments C, D, and B are not loaded until referenced.

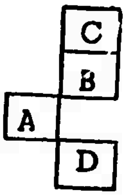
Modules may not be referenced more than once in the segment specification. The example:

SEG A, (C=B, C=D)

is illegal. The above example should be specified as:

SEG A, (C=B=D)

which represents



where C, B, and D are overlayed.

The FILE statement:

The FILE statement has the form:

FILE file-name

where "file-name" is the file to contain the load module generated by the linkage editor.

The MAP statement (optional):

The MAP statement has the form:

MAP file-name

where "file-name" indicates the file to contain the storage allocation map. A memory map will contain:

1) Names and locations of all entry points with a cross reference,

2) Name, length, and location of all common blocks,

3) Segment or subsegment number for each routine.

The BLOCK-DATA statement (optional):

The BLOCK-DATA statement has the form:

BLOCK-DATA file-name

where "file-name" indicates the file containing the block data object module.

The END statement:

The END statement has the form:

END

and indicates the end of the linkage editor control statements.

Examples

The following is an example of a linkage editor source file.

```
SEG         A, B, (C=D)
MAP         MAP
FILE        TEST
OMIT        E, F
BLOCK-DATA  BD
END
```

The commands may be in any order. In the above example, the memory allocation map is filed on MAP. The load module is filed on TEST with the external references E and F unresolved, and the block data object module is located on file BD.

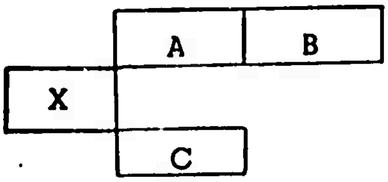
For each object module referenced in the specification portion, the following will occur.

First, the object module is relocated. This is not a straightforward process. There are two types of instruction to consider -- absolute and relocatable. For example:

```
α    SLIT (1)     A;      A = 0
     CLC (2)
     STORE (1)    $C2;
     SLIT (1)     B;      B = 2
     LIT (2)      =2;
     STORE (1)    $C2;
     JUMP         L1;     GOTO L1
B    BSS          1       END
```

where SLIT (1) A, SLIT (1) B, and JUMP L1 are relocatable instructions and the others are absolute. Also, assume that A is a member of some common block, and B is a local variable. In such an example, A must be rebound to the appropriate location in the already allocated common block. In addition, the local variable B and the label L1 are rebound relative to the programs location in memory. The absolute instructions remain unchanged.

Second , external references are resolved. If the externally referenced module exists in the same segment, the external reference is bound directly. Otherwise, the externally referenced module is accessed through a segment table. Consider the example:



where, if module A calls module B, the external reference can be bound directly. This is because modules A and B are loaded simultaneously. If, however, module A calls module C, the external reference is through the segment table causing the loading of C and making the return to module A impossible. . If module X calls module C, the external reference is through the segment table causing the loading of C and making the return to module X possible.

Third, the actual and formal parameters of external references are checked for consistent data types, and array declarations are checked for consistency. Consider the example:

```
SUBROUTINE A (I, I)
COMMON /TAB/ ARRAY (100)
            .
            .
            .
END

SUBROUTINE B
COMMON /TAB/ ARRAY (10,10)
CALL A (I, R)
.
.
.
END
```

where, I and R imply integer and real variable respectively. The linkage editor will notify the user that the data types of the actual arguments of the call to subroutine A, from subroutine B, are inconsistent with the formal parameter expected by subroutine A. In addition, the user will be notified of the discrepancy in the declaration of ARRAY. The error diagnostics are discussed in the next section.

Diagnostics

The linkage editor detects a variety of errors in the syntax and semantics of the control statements. These errors are divided into two classes -- fatal errors and non-fatal errors.

A fatal error is one from which the linkage editor cannot proceed. A load module will not be generated in such a case. A non-fatal error results in a load module being produced; however, it may contain errors.

Some of the error messages produced by the linkage editor are:

ILLEGAL MNEMONIC - STATEMENT IGNORED

An illegal mnemonic operation has been located in the control statements. Non-fatal error.

INPUT/OUTPUT ERROR

A disk input/output error has taken place. Fatal error.

SEGMENT NAME TABLE OVERFLOW

The number of segments or subsegment exceeds the length of the segment table. Fatal error.

ILLEGAL SEGMENT SYNTAX

Indicates an illegal syntax in the specification portion of a SEG statement. The statement is ignored. Non-fatal error.

ILLEGAL SYMBOL

Indicates an illegal module or file name. The statement is ignored. Non-fatal error.

SYMTAB OVERFLOW

The number of symbol table entries has exceeded the length of the symbol table. Fatal error.

EQUALS AT FIRST LEVEL

An equals (=) was detected without having first encountered a left parenthesis. The statement is ignored. Non-fatal error.

symbol_i IS NOT DISJOINT FROM symbol_j

"Symbol_i" calls "symbol_j" and they overlay each other in the same subsegment. Non-fatal error.

symbol UNDEFINED

Upon completion of the linking, the external reference "symbol" has not been defined. Non-fatal error.

symbol MULTIPLY DEFINED

The external reference "symbol" is multiply defined in the specification portion of the SEG statements. The statement is deleted. Non-fatal error.

MEMORY OVERFLOW

The main storage of the ILLIAC IV has been exceeded. Fatal error.

INCONSISTENT PARAMETERS FOR symbol

The data types between the actual and formal parameter for the routine "symbol" are inconsistent. Non-fatal error.

INCONSISTENT ARRAY DECLARATION FOR symbol

The declaration for array "symbol" is inconsistent. Non-fatal error.